Progetto di Sistemi Operativi 3

Daniele Capuano

a.a. 2010-2011

Indice

1 Descrizione generale del progetto **4**
1.1 Il *Thereminux*: descrizione generale 6
1.1.1 Configurazione software 7
1.1.2 Gestione dell'esecuzione 8
1.1.3 Gestione della sicurezza 9
1.1.4 Ambiente di esecuzione 9

2 Requisiti di progetto **10**

3 Studio di fattibilità **15**
3.1 La conversione Analogico/Digitale 15
3.1.1 Flash ADC 17
3.1.2 ADC ad approssimazioni successive 17
3.2 Lo *Universal Serial Bus* rev. 2.0 19
3.2.1 Specifiche elettriche 20
3.2.2 Modello di *Data Flow* e protocollo di comunicazione . . 20
3.2.3 Definizione di un dispositivo USB 23
3.3 I Device Driver per Linux e il kernel 2.6 27
3.3.1 I moduli del kernel Linux 29
3.3.2 Scrivere device driver per Linux 33
3.3.3 Il *Sysfs* 36
3.3.4 `udev` e la gestione dell'hotplug 37
3.3.5 I device driver e il sottosistema USB 39
3.4 Lo standard MIDI 43
3.4.1 Il protocollo software e i messaggi MIDI 43
3.5 La rappresentazione dei device MIDI in Linux: il device driver di test *minisnd* 50
3.5.1 Dettagli implementativi del driver *minisnd* 52
3.5.2 Esecuzione del driver e risultati 60
3.6 La conversione *Voltage-to-MIDI* 62
3.6.1 Algoritmi time-domain 63

3.6.2 Algoritmi frequency-domain . . . 64
3.6.3 Metodi statistici . . . 65
3.7 Tecnologie esistenti appropriate per il sistema . . . 66
3.7.1 La scheda *Beagle Board* e l'API USB Gadget del kernel Linux . . . 66
3.7.2 Il kernel *real time* . . . 70
3.7.3 Implementazioni per la *pitch detection* . . . 74

4 Cenni sulla progettazione del sistema hardware 77
4.1 Diagramma a blocchi . . . 77

5 Progettazione del sistema software 79
5.1 L'interazione con l'utente: *use cases* . . . 79
5.1.1 Specifiche dei casi d'uso . . . 81
5.2 Analisi tramite macchine a stati . . . 87
5.2.1 Dettagli sui principali scenari . . . 89
5.3 Struttura del sistema . . . 95
5.3.1 Il profilo di configurazione . . . 95
5.3.2 Script e applicazioni in *user space* . . . 103
5.3.3 Endpoint presenti sul dispositivo . . . 106
5.3.4 Comandi ioctl implementati . . . 106
5.3.5 Regole udev . . . 106
5.3.6 Il diagramma dei moduli . . . 107
5.4 Analisi delle sequenze . . . 113

6 Scelte di implementazione del sistema 125
6.1 Ambiente di sviluppo ed esecuzione . . . 125
6.1.1 Ambiente lato host . . . 126
6.1.2 Ambiente lato gadget . . . 128
6.2 I device driver sviluppati . . . 132
6.2.1 L'avvio del sistema . . . 133
6.2.2 La comunicazione USB . . . 133
6.2.3 La gestione di ALSA lato host . . . 136
6.2.4 La gestione di ALSA lato gadget e l'interfaccia a caratteri 139
6.2.5 La conversione Voltage-To-MIDI . . . 141
6.2.6 I comandi di controllo dell'esecuzione . . . 143
6.2.7 Il driver UART . . . 145
6.2.8 La gestione dell'hardware programmabile . . . 148
6.2.9 La disconnessione dello strumento . . . 148

7 Test effettuati e risultati ottenuti **150**
7.1 Test sulla trasmissione di dati audio 151
7.2 Far suonare il Thereminux 153
7.3 Test dei comandi di controllo 155
7.4 Test del MIDI-out 157

8 Conclusioni: possibili miglioramenti e integrazioni **161**

Capitolo 1

Descrizione generale del progetto

Nelle pagine che seguono viene presentata la documentazione dettagliata per il progetto dell'esame di Sistemi Operativi 3, a.a. 2010 - 2011. Il progetto consiste nella realizzazione di un piccolo strumento musicale elettronico (il *Thereminux*) interfacciato ad un device driver appositamente realizzato come modulo del kernel Linux.
In Figura 1.1 è riportato lo schema a blocchi del sistema. Come si vede e come verrà spiegato diffusamente più avanti (cfr. 1.1), il sistema è composto da una parte hardware e una software. L'hardware permette di acquisire un segnale audio analogico e di inoltrarlo, tramite usb, al sistema host, dove un device driver appositamente sviluppato lo elabora e fornisce gli strumenti di controllo remoto del dispositivo. Il sistema hardware descritto è stato realizzato utilizzando una scheda *Beagle Board* [38] (cfr. 3.7), permettendo quindi di implementare completamente l'intero protocollo di comunicazione tra host e gadget. Lo strumento utilizza il protocollo standard MIDI (*Musical Instrument Digital Interface* [17]) per la comunicazione e l'esecuzione musicale. Di seguito verrà fornita la descrizione dettagliata del progetto procedendo come segue:

- verrà data una descrizione generale del sistema software sviluppato (cfr. 1.1);
- verranno descritti i requisiti di progetto (cfr. 2);
- verrà mostrato lo studio di fattibilità effettuato prima dell'implementazione (cfr. 3) per quanto concerne il sistema software sviluppato;
- verrà fornita la documentazione di progettazione del sistema anche attraverso diagrammi UML (cfr. 5);

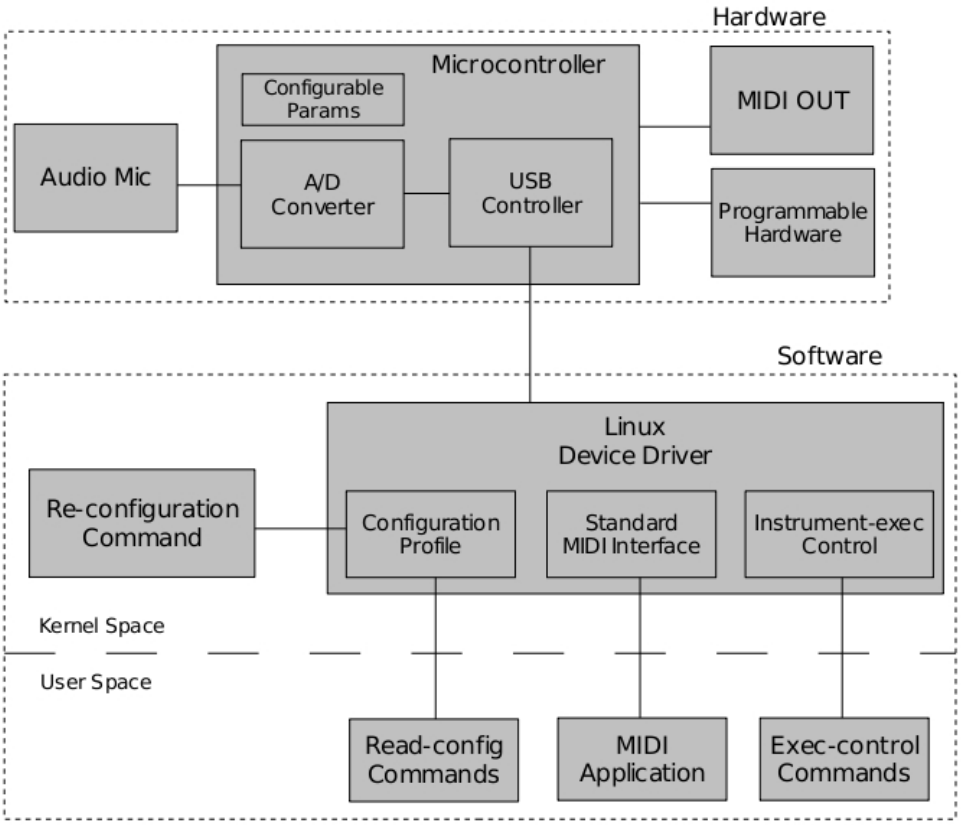

Figura 1.1: Diagramma a blocchi del *Thereminux*.

- verranno dati i dettagli tecnici dell'intero sistema software sviluppato descrivendo, per ogni modulo, le scelte a livello implementativo (cfr. 6);
- si tratteranno i test effettuati e i risultati ottenuti (cfr. 7);
- infine, si parlerà dei lavori futuri di miglioramento del sistema e si concluderà la relazione (cfr. 8).

In questo documento verrà presentato soltanto quanto concerne il sistema software del Thereminux: infatti, soprattutto a causa dell'assenza (da parte dell'autore) di una adeguata strumentazione per il debugging hardware, sono stati incontrati problemi nello sviluppo hardware e, a causa di ciò, si è preferito posticipare l'implementazione di tali dispositivi hardware (descritti nei documenti precedenti) ad un momento futuro, concentrando l'attenzione sull'intero sistema software. Così, il Theremin analogico vero e proprio in hardware, in questa fase, è stato sostituito con un semplice microfono audio collegato al convertitore messo a disposizione dalla Beagle Board. In questo modo è stato comunque possibile mettere a punto e testare l'intero progetto senza risentire dell'assenza di tale hardware.

1.1 Il *Thereminux*: descrizione generale

Come detto, il sistema hardware è interamente fornito dalla scheda Beagle Board, di cui verranno forniti alcuni dettagli più avanti (cfr. 3.7). Il sistema software che accompagna lo strumento hardware rappresenta un interprete di segnali di voltaggio che arrivano da uno strumento analogico, elettronico o elettrificato, generico. Da questo punto di vista l'idea è stata di produrre un modulo software la cui funzione principale è la conversione voltage-to-MIDI del segnale di ingresso dallo strumento, ma che mantiene una descrizione astratta della configurazione dell'hardware collegato via USB, in modo da poter essere riconfigurato per lavorare anche con strumenti hardware diversi. Ricordiamo che con l'espressione *voltage-to-MIDI* si intende qui la capacità, data una stringa binaria derivante dal campionamento di un segnale di voltaggio di input, di produrre un flusso di messaggi MIDI standard interpretabili da qualsiasi applicazione MIDI che li userà per suonarli. La descrizione dell'hardware rappresenterà una sorta di "mappa" degli elementi peculiari dello strumento collegato, come la possibilità o meno di suonare più note contemporaneamente, la presenza di periferiche di input programmabili (come pulsanti o altro), la possibilità di inviare messaggi MIDI di output attraverso lo strumento, ecc.

1.1.1 Configurazione software

Il *Thereminux* possiede la funzionalità di configurazione automatica tramite *plug and play* [20] usando la funzione di *hotplug* [19] messa a disposizione da ogni sistema operativo evoluto. Questo significa che il profilo di configurazione dello strumento (vedi Figura 1.1) viene caricato automaticamente nel sistema host al momento del suo collegamento fisico al computer. Tale *profilo di configurazione* rappresenta la descrizione astratta delle caratteristiche dello strumento e, come verrà trattato nel dettaglio più avanti (cfr. 5.3.1), è implementato come un file di configurazione in cui le diverse sezioni vengono interpretate dal driver. Il file di configurazione non è scrivibile senza opportuni privilegi ed è leggibile dagli utenti con privilegi standard solo attraverso opportuni comandi. La scrittura di un nuovo profilo può avvenire da parte di un utente privilegiato che, dopo aver modificato il file di configurazione, invoca un apposito comando che rende le modifiche effettive.
Le sezioni presenti nel profilo sono:

1. *gestione della polifonia*, ovvero la capacità dello strumento di suonare più note contemporaneamente;

2. *numero di voci* (valutato solo se lo strumento gestisce la polifonia), dice fino a quante note contemporaneamente è possibile suonare con lo strumento;

3. *playing mode*, indica se le diverse note vanno interpretate come un suono "continuo" (*glissando* [18], il modo standard in cui il Theremin suona) o come suoni separati e ben distinti (come un pianoforte, ad esempio);

4. *hardware programmabile*, è un elenco di parole chiave e indici che descrivono la presenza di un determinato elemento programmabile. Ad esempio la presenza di due pulsanti la cui funzionalità è programmabile si potrebbe indicare con le linee

   ```
   PUSHBUTTON 0
   PUSHBUTTON 1
   ```

5. *comandi di controllo dell'esecuzione*, una lista di comandi con cui è possibile controllare lo strumento durante la sua esecuzione. Ad esempio, si può configurare l'hardware per eseguire alcune trasformazioni sul valore di voltaggio da inviare via USB (uso di costanti moltiplicative o divisive, azzeramento di tutti i bit da inviare, ecc);

6. *associazioni*, ovvero una lista di coppie `(elemento-hardware, comando)` tramite le quali si definisce una associazione tra comandi di controllo dell'esecuzione dello strumento e elementi hardware programmabili presenti. In questo modo si può impostare lo strumento affinché un certo comando venga eseguito quando l'elemento hardware associato viene utilizzato. Ad esempio, un comando di controllo può essere eseguito quando viene premuto un pulsante programmabile;

7. *MIDI out*, indica se è possibile inviare messaggi MIDI di output attraverso lo strumento. Questa funzionalità può essere utilizzata quando lo strumento è collegato, oltre che al computer tramite USB, anche ad un altro dispositivo MIDI-compliant tramite cavo MIDI standard. In questo modo un'applicazione MIDI che gira in *user space* può inviare messaggi MIDI standard, tramite il device driver, allo strumento, il quale li inoltrerà al dispositivo MIDI collegato. Questa architettura può essere utilizzata quando si vuole controllare altri dispositivi MIDI tramite lo strumento, un'applicazione molto comune con la tecnologia MIDI.

1.1.2 Gestione dell'esecuzione

In runtime (ovvero una volta caricato il modulo del kernel nel sistema host), come si vede in Figura 1.1, il device driver è in grado di fornire quattro caratteristiche principali all'utente:

1. *lettura del profilo di configurazione*: la configurazione hardware corrente può essere consultata dinamicamente tramite opportuni comandi dallo *user space*;

2. *interfaccia MIDI standard*: quando l'hardware invia dei valori di voltaggio, lo strumento è in grado di generare uno stream di messaggi MIDI standard relativi a questi valori, di modo che un'applicazione in *user space* può vedere il Thereminux come un dispositivo di MIDI-input standard da cui può leggere dati. Nel caso in cui il profilo di configurazione è impostato per supportare anche il MIDI-out, il dispositivo appare ai livelli software applicativi anche come un'interfaccia di MIDI-output standard su cui poter inviare messaggi. Quest'ultima funzionalità può variare in runtime: ad esempio, un dispositivo MIDI esterno potrebbe venire collegato o scollegato al Thereminux quando questo è già in esecuzione. Per questo è stato implementato un apposito comando, eseguibile da un utente con privilegi standard, che permette di attivare e disattivare dinamicamente questa funzione.

3. *controllo di esecuzione dello strumento*: i vari comandi di controllo, che vengono elencati all'interno del profilo di configurazione, oltre a poter essere associati agli elementi hardware programmabili, possono anche essere invocati dal software in esecuzione sul host. In particolare, sono stati messi a disposizione dei comandi, eseguibili con privilegi standard, con i quali è possibile invocare queste funzioni di controllo. La lista dei comandi di controllo disponibili può essere ottenuta tramite uno dei comandi che permettono di leggere il profilo di configurazione;

4. *riconfigurazione*: un utente privilegiato può cambiare dinamicamente il profilo di configurazione e registrare le tali modifiche sul dispositivo tramite un apposito comando.

1.1.3 Gestione della sicurezza

Il device driver implementa alcuni meccanismi di sicurezza. In particolare, si effettuano controlli su eventuali parametri passati ai comandi di controllo e si verifica, ad ogni tentativo di riconfigurazione dell'hardware, che il profilo che si cerca di impostare sia ben formato e corretto. Infine, poiché il Thereminux appare alle applicazioni come un normale dispositivo MIDI compatibile con gli ambienti audio standard per linux (come *ALSA* [32]), i messaggi MIDI possono essere inoltrati sulla porta di MIDI-out del Thereminux soltanto attraverso la sua interfaccia MIDI virtuale e, dunque, soltanto da applicazioni MIDI standard. Per questo, non viene effettuato un controllo sui messaggi di MIDI-out inoltrati, in quanto chi li genera è considerato affidabile.

1.1.4 Ambiente di esecuzione

Per ragioni di efficienza, l'applicazione è stata implementata cercando di rispondere al requisito non funzionale di bassa latenza. Questo per evitare (o comunque minimizzare) il ritardo percepibile tra un input sottoposto dal lato hardware e l'inizio della nota relativa. Per questo il kernel sul sistema host è stato riconfigurato per gestire al meglio applicazioni real-time.

Capitolo 2

Requisiti di progetto

Di seguito, è riassunto schematicamente l'elenco completo dei requisiti che il sistema *Thereminux* soddisfa. L'elenco è suddiviso in tre categorie:

1. Requisiti hardware (**HW**);
2. Requisiti software (**SW**);
3. Requisiti di sicurezza (**SEC**).

Ad ogni requisito è stata associata una stringa identificativa che ne indica:

- La tipologia primaria (**HW**, **SW**, **SEC**);
- La tipologia secondaria (**F**unzionale, **N**on **F**unzionale);
- L'identificativo numerico progressivo.

Ad esempio, un requisito software funzionale sarà associato all'identificativo `RSW_F.`*`X`*, dove *X* rappresenta un dentificativo numerico.
Inoltre, ad ogni requisito è associato un livello di priorità. Questa informazione è utilizzata come riferimento nelle varie fasi di sviluppo per stabilire:

- quali funzionalità del sistema sono più importanti e devono quindi essere necessariamente presenti fin dall'inizio (*MustHave*);
- quali funzionalità potrebbero essere implementate solo parzialmente all'inizio e poi perfezionate in un successivo aggiornamento del software (*ShouldHave*);
- quali funzionalità, infine, possono essere implementate direttamente in successivo aggiornamento del software (*MayHave*).

Chiaramente, come già detto in apertura della relazione, poiché il sistema hardware è stato sviluppato soltanto per quanto riguarda le funzionalità messe a disposizione dalla Beagle Board, alcuni dei requisiti hardware (come l'implementazione in hardware del Theremin analogico) vengono mostrati con la priorità *MayHave*.

Requisiti hardware			
ID	**Descrizione**	**Tipo**	**Priorità**
RHW_F.1	Il sistema dovrà essere costruito in prima istanza come un Theremin analogico	Funzionale	MayHave
RHW_F.2	Il sistema dovrà convertire il segnale analogico prodotto in una stringa binaria tramite un modulo A/D	Funzionale	MustHave
RHW_F.3	Il sistema hardware dovrà comunicare con il pc tramite interfaccia standard USB	Funzionale	MustHave
RHW_F.4	Lo strumento potrà disporre di un collegamento di MIDI-OUT per il controllo di altri strumenti MIDI-compliant	Funzionale	ShouldHave
RHW_F.5	Lo strumento potrà disporre di elementi hardware programmabili	Funzionale	ShouldHave
RHW_F.6	Lo strumento dovrà garantire i meccanismi hardware per il funzionamento in *plug and play* (tramite hotplug)	Funzionale	ShouldHave
RHW_F.7	Lo strumento dovrà disporre di un apposito microcontroller per la gestione delle funzionalità programmabili in hardware e controllabili da software	Funzionale	MustHave

Requisiti software			
ID	**Descrizione**	**Tipo**	**Priorità**
RSW_F.1	Lo strumento dovrà garantire i meccanismi software per il funzionamento in *plug and play* (tramite hotplug)	Funzionale	MustHave
RSW_F.2	Il sistema, attraverso il meccanismo plug and play, dovrà impostare automaticamente il profilo di configurazione	Funzionale	MustHave
RSW_F.3	Il device driver dovrà acquisire le stringhe binarie di ingresso ed effettuare una conversione voltage-to-MIDI di queste	Funzionale	MustHave
RSW_F.4	Il device driver dovrà fornire un'interfaccia MIDI standard verso il livello applicativo	Funzionale	MustHave
RSW_F.5	Lo strumento dovrà essere configurato tramite un *profilo di configurazione* che rappresenta la descrizione astratta delle caratteristiche dello strumento	Funzionale	MustHave
RSW_F.6	Il profilo di configurazione dovrà contenere il campo di *gestione della polifonia* in cui si specifica se lo strumento ha la possibilità di suonare più note contemporaneamente	Funzionale	MustHave
RSW_F.7	Il profilo di configurazione dovrà contenere il campo *numero voci*, valutato soltanto se lo strumento gestisce la polifonia	Funzionale	MustHave
RSW_F.8	Il profilo di configurazione dovrà contenere il campo *playing mode*. Questo specifica se le note vanno suonate in maniera continua o discreta	Funzionale	MustHave
RSW_F.9	Il profilo di configurazione dovrà contenere il campo *hardware programmabile*, ovvero un elenco di parole chiave e indici che descrivono i diversi elementi hardware programmabili (ad esempio, push button) presenti nello strumento	Funzionale	MustHave

`RSW_F.10`	Il profilo di configurazione dovrà contenere il campo *comandi di controllo dell'esecuzione*. Questo specifica un elenco di comandi che è possibile eseguire dal lato software per controllare i parametri di esecuzione dell'hardware	Funzionale	MustHave
`RSW_F.11`	Il profilo di configurazione dovrà contenere il campo *associazioni*. Questo specifica una serie di coppie `(elemento-hardware, comando)` con cui si impone che un certo comando di controllo dell'esecuzione venga eseguito quando si attiva lo `elemento-hardware`	Funzionale	MustHave
`RSW_F.12`	Il profilo di configurazione dovrà contenere il campo *MIDI out*, che indica se è possibile inviare dati MIDI di output attraverso lo strumento	Funzionale	MustHave
`RSW_F.13`	Il sistema dovrà mettere a disposizione un comando per leggere il profilo di configurazione	Funzionale	MustHave
`RSW_F.14`	Il sistema dovrà mettere a disposizione un comando per modificare in run-time la possibilità di inviare messaggi MIDI OUT attraverso lo strumento	Funzionale	MustHave
`RSW_F.15`	Il sistema dovrà mettere a disposizione dei comandi per invocare dal lato software i comandi di controllo dell'esecuzione	Funzionale	MustHave
`RSW_F.16`	Il sistema dovrà mettere a disposizione un comando per leggere l'elenco dei comandi di controllo dell'esecuzione	Funzionale	MustHave
`RSW_F.17`	Il sistema dovrà mettere a disposizione un comando per rendere effettive le modifiche al profilo di configurazione effettuate	Funzionale	MustHave
`RSW_NF.1`	Il device driver andrà implementato per il kernel Linux 2.6	Non Funzionale	ShouldHave
`RSW_NF.2`	Il kernel dovrà essere ottimizzato per lavorare in real-time	Non Funzionale	MustHave
`RSW_NF.3`	Il sistema dovrà avere una bassa latenza tra gli input e gli output	Non Funzionale	ShouldHave

Requisiti di sicurezza			
ID	**Descrizione**	**Tipo**	**Priorità**
RSEC_NF.1	La lettura del profilo di configurazione dovrà poter essere effettuata con privilegi standard	Non Funzionale	MustHave
RSEC_NF.2	La lettura del profilo di configurazione, da parte di utenti con privilegi standard, dovrà poter essere effettuata solo attraverso i comandi messi a disposizione dal sistema	Non Funzionale	MustHave
RSEC_NF.3	Il sistema dovrà permettere a utenti con privilegi di amministrazione di leggere e modificare con qualsiasi editor il file del profilo di configurazione	Non Funzionale	MustHave
RSEC_NF.4	Il comando presente nel sistema per rendere effettive le modifiche al profilo di configurazione dovrà poter essere invocato soltanto da utenti con privilegi di amministrazione	Non Funzionale	MustHave
RSEC_NF.5	La modifica della presenza o meno della funzionalità di MIDI OUT, tramite apposito comando, dovrà poter essere effettuata con privilegi standard	Non Funzionale	MustHave
RSEC_NF.6	L'invocazione dei comandi di controllo dell'esecuzione dovrà poter essere effettuata con privilegi standard	Non Funzionale	MustHave
RSEC_NF.7	La lettura dell'elenco dei comandi di controllo dell'esecuzione dovrà poter essere effettuata con privilegi standard	Non Funzionale	MustHave
RSEC_NF.8	Il sistema dovrà effettuare dei controlli su tutti gli eventuali parametri passati ai comandi di controllo dell'esecuzione	Non Funzionale	MustHave
RSEC_NF.9	Il sistema dovrà verificare, ad ogni tentativo di riconfigurazione, che il profilo di configurazione sia ben formato e corretto	Non Funzionale	MustHave

Capitolo 3

Studio di fattibilità

In questo capitolo verranno analizzate tutte le questioni emerse dall'analisi dei requisiti (cfr. Capitolo 2), per mostrare l'esistenza di tecnologie appropriate per lo sviluppo degli aspetti hardware/software del sistema, studiarne le caratteristiche e descrivere le soluzioni trovate.
Per la realizzazione del sistema ci sono diverse questioni da analizzare:

1. la conversione A/D del segnale analogico in uscita del dispositivo audio di input (requisito RHW_F.2): vedi Sezione 3.1;
2. l'utilizzo dello standard USB per il collegamento verso il pc (requisito RHW_F.3): vedi Sezione 3.2;
3. lo sviluppo di un device driver USB per Linux, kernel 2.6 (requisito RSW_NF.1): vedi Sezione 3.3;
4. lo standard MIDI per la gestione del flusso di dati di input e output (requisito RSW_F.4): vedi Sezione 3.4;
5. la rappresentazione dei device MIDI in Linux (requisito RSW_F.4): vedi Sezione 3.5;
6. la conversione voltage-to-MIDI (requisito RSW_F.3): vedi Sezione 3.6.

Studiamo nel dettaglio ognuno di questi punti, al fine di analizzare le varie caratteristiche necessarie per soddisfare i requisiti del sistema e arrivare (cfr. 3.7) alle tecnologie che meglio si adattano a ciò che emerge da questa analisi.

3.1 La conversione Analogico/Digitale

Il meccanismo di conversione di un segnale analogico in una stringa digitale consiste nel trovare un numero digitale proporzionale all'ampiezza del segnale

analogico di input [1]. È una tecnica utilizzata ogni qual volta c'è la necessità di controllare con sistemi digitali un esperimento o un processo che vede coinvolte quantità analogiche (livelli continui di voltaggio o di corrente). I parametri fondamentali nella scelta di un ADC appropriato ad una specifica applicazione sono [22]:

- *Risoluzione*: numero di valori discreti che il convertitore può produrre sul range di valori analogici. Questo parametro viene espresso in bit: ad esempio, un ADC con risoluzione di 8 bit può produrre fino a $2^8 = 256$ valori discreti. In alternativa, attraverso il numero di bit utilizzati, si può esprimere la risoluzione del convertitore come la dimensione (in volt) dell'i-simo intervallo in cui è possibile dividere il valore complessivo di voltaggio del segnale di input. In altre parole si può scrivere
$$Q = \frac{V_{refHI} - V_{refLOW}}{2^M - 1}$$
dove V_{refHI} e V_{refLOW} sono, rispettivamente, il valore massimo e minimo di voltaggio che il segnale può raggiungere e M è il numero di bit del convertitore;

- *Frequenza di campionamento*: poiché il segnale è realmente ricostruito, a partire dall'insieme dei punti discreti ottenuti con il campionamento, attraverso formule di interpolazione, è necessario essere sicuri che i campioni siano presi abbastanza frequentemente da riuscire a ricostruire con una fedeltà accettabile il segnale originale. Il teorema di *Shannon-Nyquist* [21], a questo proposito, afferma che, per ricostruire una qualsiasi forma d'onda, è necessario avere a disposizione una frequenza di campionamento almeno uguale al doppio della massima frequenza presente nel segnale.

- *Accuratezza*: si riferisce alla "vicinanza" del numero digitale trovato rispetto al valore reale di voltaggio del segnale. Fonti di errore in questo senso sono:

  - Errori di quantizzazione, inevitabili a causa del numero finito di bit che vengono usati per la conversione: in generale, dati n bit, è possibile esprimere solo fino a 2^n valori binari. Ma, se prendiamo l'intervallo compreso tra 2^{i-1} e 2^i, con $0 < i \leq n$, questo conterrà k valori di voltaggio ($k > 1$) ma verrà rappresentato con una sola stringa binaria;
  - Non linearità. La maggior parte degli ADC vengono detti *lineari*, ovvero tra il valore di voltaggio di input e il valore binario di

output c'è una relazione lineare. Quando questo non avviene c'è un errore di non linearità, che riduce la quantità di valori che può produrre la conversione;

– Errori di apertura, ovvero un'incertezza nel campionamento effettuato quando il segnale è variabile nel tempo. Qui, se il campionamento non è effettuato con una cadenza temporale esatta (a causa, ad esempio, dell'utilizzo di clock esterni), si possono provocare errori di accuratezza che possono essere tanto maggiori quanto più aumenta la frequenza del segnale e la sua ampiezza.

Nel dominio del progetto qui presentato, i primi due parametri (risoluzione e frequenza di campionamento) sono quelli di maggior interesse: in generale infatti, nell'ambito dei convertitori audio, questi vengono catalogati insieme come "risoluzione" del convertitore [23]. Soprattutto, per coprire tutto il range delle frequenze udibili, teniamo a mente il teorema di Nyquist: visto che le frequenza audio vanno dai 20 Hz ai 20 KHz [24], dobbiamo avere a disposizione una frequenza di campionamento di almeno 40 KHz. Non a caso, nei sistemi audio, si usa la frequenza standard di 44.1 KHz. Per quanto riguarda il numero di bit, solitamente i convertitori audio lavorano a 8, 16 o 24 bit, anche se quelli a 8 bit risultano un po' datati.
Per una scelta appropriata del convertitore da usare per l'applicazione qui esposta, presentiamo infine le due principali tipologie di ADC: i *flash ADC* e i convertitori che lavorano per *approssimazioni successive*.

3.1.1 Flash ADC

In questa configurazione, mostrata in Figura 3.1, il segnale di input è inviato in parallelo a una serie di comparatori, di cui ognuno confronta il segnale con la propria tensione di riferimento (regolata attraverso una serie di partitori di tensione): solo un comparatore, quindi, produrrà un output, che verrà inviato ad un encoder digitale che genererà il codice per quella tensione.
Questo è il tipo più veloce di ADC, anche se è molto costoso e tende a produrre dei *glitch* sul segnale di output.

3.1.2 ADC ad approssimazioni successive

La Figura 3.2 mostra lo schema di un convertitore AD ad approssimazioni successive.
Il funzionamento di questo convertitore è molto semplice: l'idea è quella di effettuare vari "tentativi" di codifiche binarie in cui, ad ogni tentativo, la stringa è convertita in analogico tramite un modulo DAC (*Digital-Analog*

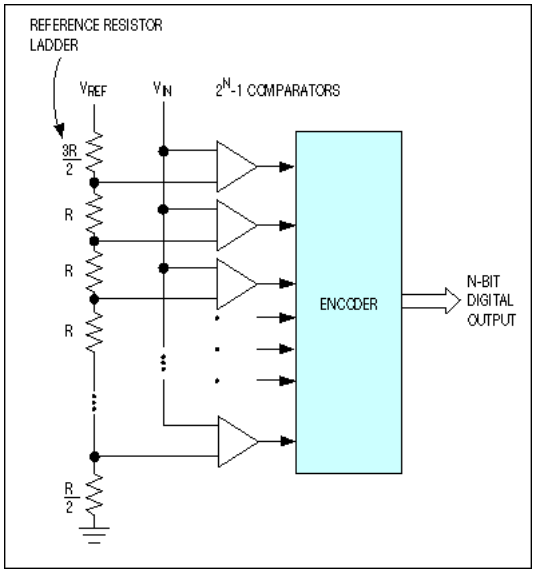

Figura 3.1: Schema di un Flash ADC. Fonte [25]

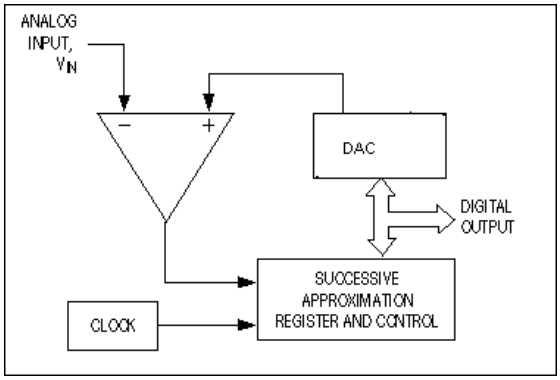

Figura 3.2: Schema di un ADC ad approssimazioni successive. Fonte [25]

Converter) interno. I confronti vengono effettuati a partire dal voltaggio di riferimento, che rappresenta il massimo voltaggio che il segnale può raggiungere. Ad esempio [22], supponiamo che il voltaggio di input sia di 60 V e che il voltaggio di riferimento sia 100 V. Per prima cosa verrà confrontato 60 V con 50 V (che è la metà del voltaggio di riferimento, e corrisponde alla stringa binaria `100...0`). Il comparatore produrrà il valore `1` (visto che 60 V $>$ 50 V) e questo sarà memorizzato come MSB (*Most Significant Bit*). Nella seconda iterazione del processo, 60 V è confrontato con 75 V (il valore mediano tra 50 V e 100 V, in binario `1100...0`). Poiché 60 V $<$ 75 V, il comparatore produrrà uno `0` e, quindi, il secondo bit del risultato verrà posto a `0`. Al terzo ciclo, quindi, il voltaggio di input è confrontato con 62.5 V (il valore mediano tra 50 V e 75 V, in binario `10100...0`) e anche qui verrà memorizzato uno `0`. E così via. In pratica, si esegue una ricerca binaria sul valore di voltaggio di input per trovare la codifica che approssima meglio.
Naturalmente, per ogni codifica, c'è bisogno di effettuare n passi, con n uguale al numero di bit di risoluzione dell'ADC. Questo tipo di convertitori necessitano di avere una frequenza di clock uguale alla frequenza di campionamento desiderata per il numero di bit di risoluzione:

$$f_{clock} = f_{sampling} \cdot n_{bit}$$

Ad esempio, nel nostro caso, per un campionamento a 44.1 KHz e 16 bit di risoluzione, c'è bisogno di una frequenza di clock di cira 705 KHz.
Questi convertitori sono molto usati per la loro accuratezza e il costo relativamente basso.

3.2 Lo *Universal Serial Bus* rev. 2.0

Lo *Universal Serial Bus* [2] è un bus che permette lo scambio di dati tra un *Host* centrale e una serie di dispositivi (*Device*) che possono essere collegati e scollegati dinamicamente anche durante l'esecuzione delle operazioni. I dispositivi possono essere di due tipi:

- *Hub*, che permettono di collegare nuovi client all'Host centrale;
- *Funzioni*, i client veri e propri, che forniscono funzionalità aggiuntive al sistema.

Lo scambio dei dati può avvenire in diverse modalità:

- *High speed*, con un bit rate fino a 480 Mb/s (solo USB 2.0);
- *Full speed*, bit rate fino a 12 Mb/s;

- *Low speed*, con un bit rate fino a 1.5 Mb/s

L'esistenza delle ultime due velocità di scambio dei dati ha senso principalmente per un fattore di retrocompatibilità con le versioni precedenti dell'USB.

3.2.1 Specifiche elettriche

La Figura 3.3 mostra lo schema del cavo USB.

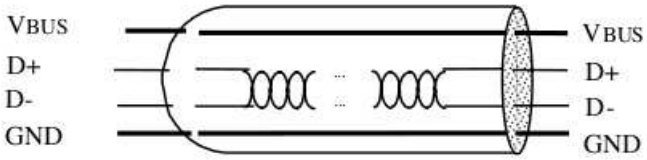

Figura 3.3: Cavo USB. Fonte [2]

Il segnale dei dati viaggia sul cavo attraverso due pin (D+ e D-), mentre il segnale di voltaggio di riferimento (di 5 V) è portato dal pin V_{BUS} (in questo modo i dispositivi possono ottenere l'alimentazione dallo stesso cavo USB) e quello di massa dal pin GND. Inoltre, il clock viene inviato insieme ai dati attraverso un pacchetto di SYNC, in modo che le comunicazioni distribuite possano essere sincronizzate.

3.2.2 Modello di *Data Flow* e protocollo di comunicazione

L'idea alla base dell'USB è di offrire una generalità tale da poter sfruttare questo bus per le diverse esigenze di comunicazione di molti dispositivi. Per questo i flussi di dati sono gestiti a livello logico attraverso l'uso di *pipe* unidirezionali, gestite in modo indipendente una rispetto all'altra, sul cui numero e tipo l'host e il dispositivo possono accordarsi liberamente in base alle esigenze del dispositivo. La Figura 3.4 mostra il dettaglio di come questi flussi sono strutturati.
Come si vede possiamo riconoscere tre livelli di stratificazione:

1. il livello *fisico*: l'ultimo dal basso in Figura 3.4, dove viene implementato il protocollo hardware e i dati vengono scambiati attraverso il cavo USB descritto precedentemente (cfr. 3.2.1);

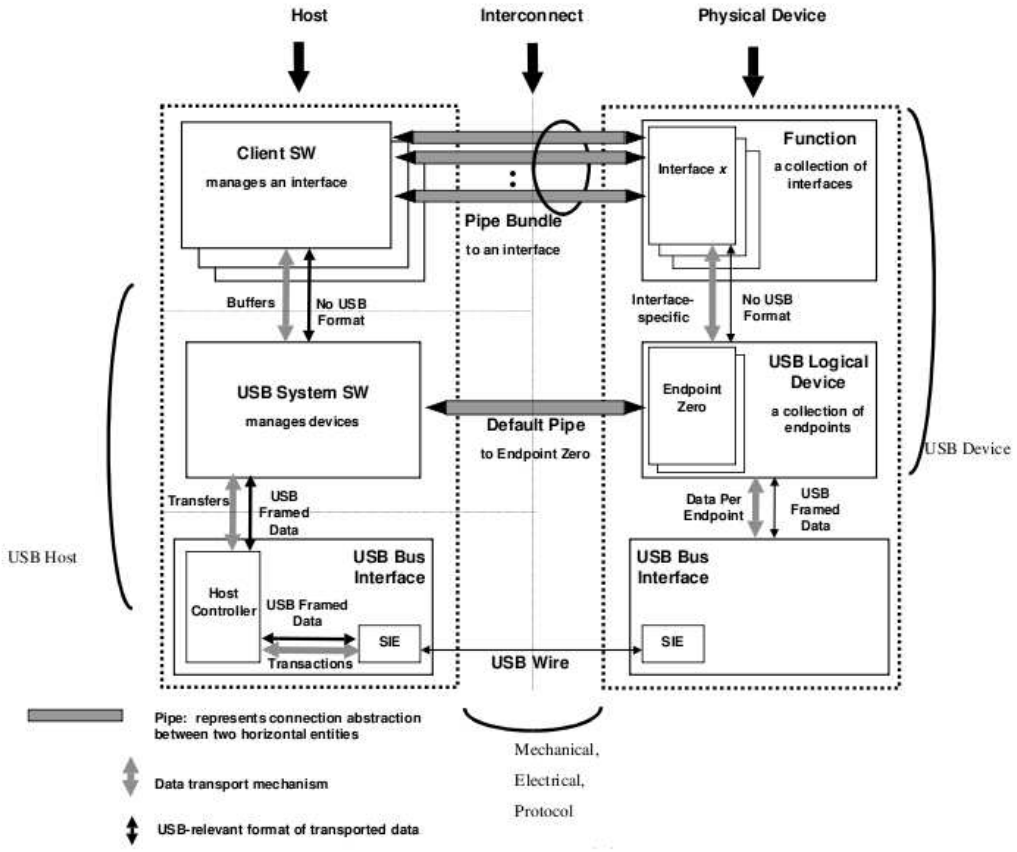

Figura 3.4: Modello di comunicazione stratificato tra Host e Device USB. Fonte [2]

2. il livello di *dispositivo logico*: è un insieme di *endpoint*, che sono definiti come terminali identificati univocamente, lato dispositivo, dei flussi di comunicazione tra host e dispositivo. Ogni dispositivo viene associato, nel momento del suo collegamento fisico, ad un indirizzo univoco tra tutti i dispositivi collegati all'host; inoltre, ogni endpoint ha un *numero di endpoint*, che lo identifica all'interno del dispositivo. Così, l'identificazione univoca dell'endpoint è fornita da una terna composta da:

   (a) indirizzo del dispositivo logico;

   (b) numero di endpoint;

   (c) direzione del flusso di comunicazione: infatti, come già detto, ogni flusso può essere pensato come una pipe unidirezionale: in questo

senso possiamo parlare di flussi di *input* (dal dispositivo all'host) e di *output* (dall'host al dispositivo).

Ogni dispositivo logico dispone, inoltre, di un endpoint di controllo, ovvero l'*endpoint numero 0* o *Default Control Pipe*. Questo è utilizzato, lato host, dall'USB System Software per effettuare tutte le operazioni di setup del dispositivo al momento del suo collegamento fisico. Tale USB System Software è composto da tutte le funzionalità messe a disposizione dal sistema operativo dell'host per gestire lo USB a basso livello, fornendo così al programmatore un'interfaccia sulla quale basare lo sviluppo del driver specifico del dispositivo;

3. il livello di *funzione*: ogni funzione è composta da un insieme di *interfacce*, ognuna delle quali raggruppa l'insieme di endpoint che la implementano. Un dispositivo che implementa più di una interfaccia viene detto *composite device*. A questo livello la comunicazione logica avviene tra *client software* (lato host), ovvero il device driver vero e proprio, e una *interfaccia* sul dispositivo. Questa astrazione permette al software sull'host di concentrarsi sul richiamare le funzionalità descritte dall'interfaccia del dispositivo, utilizzando quindi un particolare *bundle* di endpoint associati a quella interfaccia.

Per chiarire meglio, dunque, il dispositivo definisce una serie di endpoint, ognuno con le sue caratteristiche (identificativo, direzione della comunicazione). Oltre a questo, definisce una o più interfacce con le funzioni specifiche fornite dal dispositivo. La comunicazione logica tra il software sull'host e un endpoint sul dispositivo avviene attraverso una *pipe USB*, ovvero un flusso di comunicazione che permette di accedere ad un certo endpoint dal software sull'host, definendo le caratteristiche di questa comunicazione. Le più importanti tra queste caratteristiche sono:

- la bandwidth richiesta per i trasferimenti;
- i limiti sulla dimensione dei pacchetti trasferiti;
- le caratteristiche dell'endpoint associato alla pipe (ad esempio la direzione del flusso);
- il *tipo di trasferimento*. In particolare, lo USB definisce quattro tipi di trasferimento di dati:
  - *Control*: trasferimenti di tipo non peridico, richiesti dall'host, per funzioni di controllo del dispositivo;

- *Isochronous*: comunicazioni di flussi continui di informazioni che hanno dei requisiti stringenti di tempo (real-time);
- *Interrupt*: trasferimenti periodici di pacchetti, di solito piuttosto piccoli, di informazioni che vanno gestite con una certa "urgenza";
- *Bulk*: trasferimenti non periodici di grandi quantità di dati che non hanno particolari requisiti di tempo, ma possono anche subire ritardi senza per questo intaccare la qualità della comunicazione.

Quindi, ad *attach time*, l'hub principale dell'host rileva un nuovo dispositivo consultando un opportuno *status bit*; quando il dispositivo viene rilevato, inizia la fase di configurazione, in cui, prima di tutto, l'host assegna un indirizzo univoco al dispositivo; successivamente, tramite la Default Control Pipe, l'host richiede al dispositivo tutte le informazioni relative alla struttura dei flussi di comunicazione necessari al dispositivo stesso (interfacce, endpoint, ecc). Successivamente il device driver sull'host alloca tutte le strutture per gestire questi flussi di comunicazione.
Fatto questo, il protocollo è sostanzialmente basato sul *polling*, con l'host che inizia periodicamente ogni trasmissione, inviando i dati (nel caso di flussi di output) o chiedendo al dispositivo se ce ne sono di nuovi (nel caso di flussi di input). Queste operazioni sono sempre iniziate, da parte dell'host, inviando dei pacchetti di *token* che specificano una richiesta per una trasmissione indirizzata ad un certo endpoint sul dispositivo. Quest'ultimo risponde a questa richiesta inviando un *pacchetto di handshake* che può confermare la disponibilità a iniziare la trasmissione (pacchetto `ACK`) oppure, se è impossibilitato a ricevere o non ha nulla da inviare, declinare la richiesta (pacchetto `NAK`). Nel caso in cui l'host effettui una richiesta per un'operazione non valida o non supportata dal dispositivo, questo può inviare il pacchetto di handshaking `STALL`, dopo il quale il dispositivo si trova in uno stato indefinito.

3.2.3 Definizione di un dispositivo USB

Sostanzialmente, dunque, per quanto spiegato nel paragrafo precedente (cfr. 3.2.2), l'host si occupa di rilevare un nuovo dispositivo USB collegato e inviare le richieste per la sua configurazione e per effettuare lo scambio dei dati. Queste funzionalità, all'interno dell'host, sono già implementate a livello di sistema operativo in maniera trasparente all'utente, al quale viene fornita una API per sviluppare il device driver specifico per il dispositivo senza dover fare i conti con il protocollo USB vero e proprio (per i dettagli cfr. 3.3). Inoltre, non verranno in questa sede considerati i dettagli per l'implementazione hardware dell'USB sul dispositivo, in quanto (cfr. 3.7.1), come già

accennato in apertura del lavoro, la scheda hardware utilizzata fornisce, oltre ad un microprocessore programmabile integrato, tutta la circuiteria necessaria all'implementazione hardware del protocollo USB.
Ciò che, dunque, rimane comunque necessario è impostare il firmware del dispositivo hardware perché sia in grado di rispondere correttamente alle richieste che il sistema operativo gli invierà una volta effettuato il collegamento fisico e fornita l'alimentazione. La specifica USB [3] fornisce tutti i dettagli su come queste richieste devono essere gestite da parte del dispositivo.
Ogni richiesta da parte dell'host arriva al dispositivo attraverso la sua Default Control Pipe (endpoint 0). Questa richiesta è formata da diversi campi, come mostrato in Figura 3.5
Il campo *bmRequestType*, una bitmap, specifica le caratteristiche generali del trasferimento, cioè la direzione, il tipo di richiesta (standard, riferita alla classe del dispositivo, ecc) e l'oggetto a cui fa riferimento (dispositivo, interfaccia, endpoint); *bRequest* è un codice che identifica la richiesta vera e propria (per una lista delle richieste standard che tutti i dispositivi USB devono gestire si faccia riferimento alla Figura 3.6). I successivi due campi (*wValue* e *wIndex*) variano a seconda del tipo di richiesta, ma sono utilizzati per passare dei parametri al dispositivo assieme alla richiesta. Infine *wLength* specifica quanti byte di dati ci si aspetta come risposta.
Come si vede in Figura 3.6, le richieste standard che ogni dispositivo USB deve essere in grado di gestire sono molte, e mettono in evidenza altri due fondamentali concetti che devono essere implementati dal dispositivo: le configurazioni e i descrittori. Ogni dispositivo USB, infatti, può avere diverse *configurazioni*: fondamentalmente una configurazione è un insieme di interfacce implementate dal dispositivo e l'insieme delle *impostazioni alternative* di ogni interfaccia. Se, come detto, ogni interfaccia è associata ad un insieme di endpoint che la implementano, usando un'impostazione alternativa per quell'interfaccia essa può variare dinamicamente il numero e il tipo di endpoint a cui è associata. Per quanto sicuramente permette la gestione di dispositivi anche molto complessi, questa caratteristica è comunque raramente utilizzata.
I *descrittori*, invece, sono le strutture dati con cui effettivamente vengono specificati tutti gli attributi del dispositivo. Tra questi attributi abbiamo la specifica del dispositivo (numero di configurazioni, numero di interfacce, numero di endpoint, *vendor*, ecc), la specifica delle configurazioni, delle interfacce e degli endpoint per il dispositivo, con inoltre una serie di descrittori "stringa" utilizzati per memorizzare e inviare delle informazioni leggibili e presentabili agli utenti. Infine, nel caso di dispositivi che possono essere eseguiti in *high-speed*, è necessario specificare sia una configurazione di compatibilità per l'esecuzione in *full-speed* (dichiarando descrittori di entrambi i tipi) e,

Offset	Field	Size	Value	Description
0	*bmRequestType*	1	Bitmap	Characteristics of request: D7: Data transfer direction 0 = Host-to-device 1 = Device-to-host D6...5: Type 0 = Standard 1 = Class 2 = Vendor 3 = Reserved D4...0: Recipient 0 = Device 1 = Interface 2 = Endpoint 3 = Other 4...31 = Reserved
1	*bRequest*	1	Value	Specific request
2	*wValue*	2	Value	Word-sized field that varies according to request
4	*wIndex*	2	Index or Offset	Word-sized field that varies according to request; typically used to pass an index or offset
6	*wLength*	2	Count	Number of bytes to transfer if there is a Data stage

Figura 3.5: Struttura di una richiesta USB. Fonte [3]

inoltre, un *Device Qualifier* che permette al dispositivo di descrivere come si può comportare ad entrambe le velocità, così che l'host possa scegliere la migliore per l'applicazione da eseguire.
Quello che, dunque, fa un dispositivo alla ricezione di una richiesta, ad esempio, come la `GetDescriptor()` è andare a recuperare il descrittore specificato come argomento della richiesta e inviarlo all'host nella risposta. La struttura di ogni richiesta è sempre come quella mostrata in Figura 3.5 e, quindi, gli attributi vengono inviati utilizzando i campi *wValue* e/o *wIndex*.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00000000B 00000001B 00000010B	CLEAR_FEATURE	Feature Selector	Zero Interface Endpoint	Zero	None
10000000B	GET_CONFIGURATION	Zero	Zero	One	Configuration Value
10000000B	GET_DESCRIPTOR	Descriptor Type and Descriptor Index	Zero or Language ID	Descriptor Length	Descriptor
10000001B	GET_INTERFACE	Zero	Interface	One	Alternate Interface
10000000B 10000001B 10000010B	GET_STATUS	Zero	Zero Interface Endpoint	Two	Device, Interface, or Endpoint Status
00000000B	SET_ADDRESS	Device Address	Zero	Zero	None
00000000B	SET_CONFIGURATION	Configuration Value	Zero	Zero	None
00000000B	SET_DESCRIPTOR	Descriptor Type and Descriptor Index	Zero or Language ID	Descriptor Length	Descriptor
00000000B 00000001B 00000010B	SET_FEATURE	Feature Selector	Zero Interface Endpoint	Zero	None
00000001B	SET_INTERFACE	Alternate Setting	Interface	Zero	None
10000010B	SYNCH_FRAME	Zero	Endpoint	Two	Frame Number

Figura 3.6: Elenco delle richieste standard USB. Fonte [3]

3.3 I Device Driver per Linux e il kernel 2.6

Consideriamo ora lo studio degli aspetti relativi al kernel Linux. Il materiale presentato in questa sezione si deve principalmente alla consultazione di [4] e [5]. La trattazione, inoltre, non sarà rigorosa da un punto di vista dell'implementazione delle varie funzioni descritte, ma sarà finalizzata alla descrizione delle caratteristiche peculiari della programmazione di device driver (e moduli del kernel in generale) e dei punti chiave nella progettazione e programmazione di tali moduli.
Come si può vedere in Figura 3.7 il kernel Linux è un sistema molto complesso e comprendente numerose componenti. I tipi di kernel esistenti sono sostanzialmente due: *microkernel* e *monolitico* [5]. Nel primo caso, soltan-

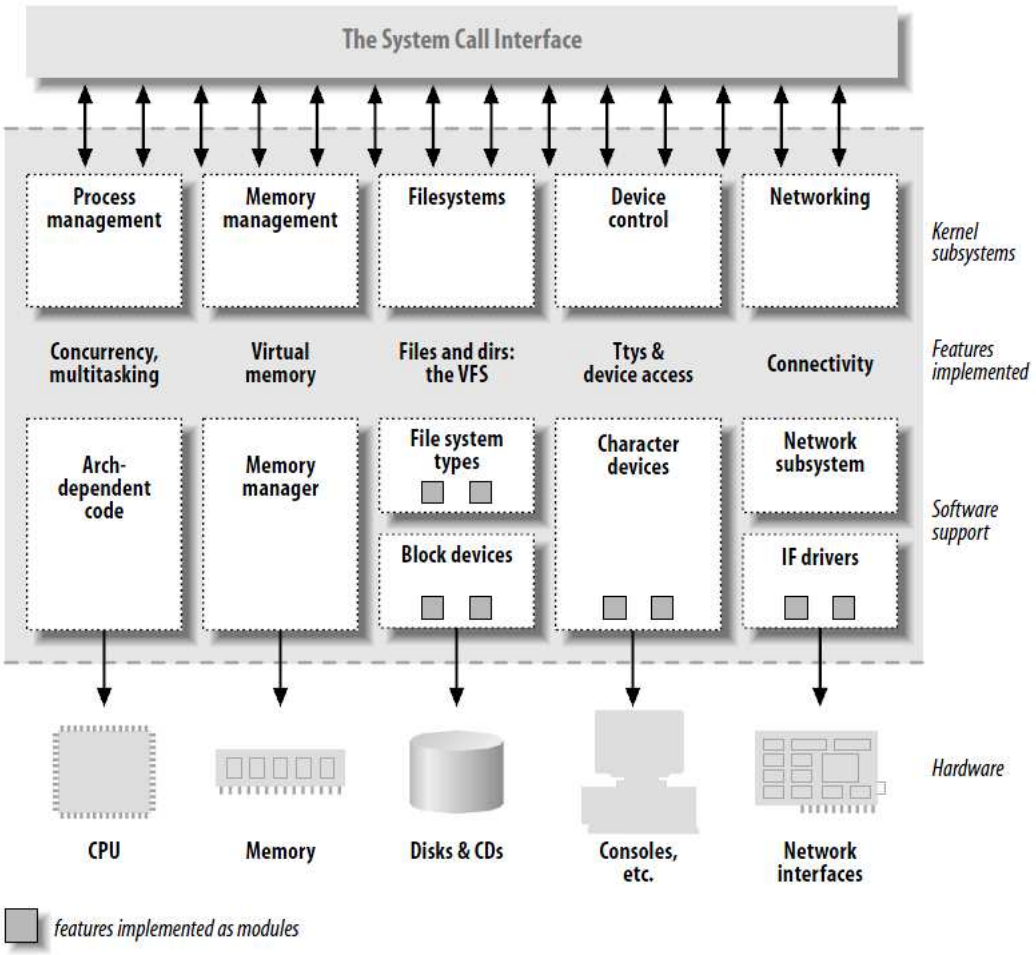

Figura 3.7: Vista del kernel linux. Fonte [4]

to una minima porzione delle operazioni è implementata nell'eseguibile del kernel, ma la maggior parte delle funzionalità è fornita da *moduli* dinamici. Questo permette una maggiore dinamicità e versatilità del codice a scapito, però, dell'efficienza visto che i vari moduli devono necessariamente comunicare tra loro attraverso una qualche forma di messaggi. Nel secondo caso, invece, il kernel è implementato come un unico blocco di codice che contiene tutte le funzionalità. Questo approccio, benché manchevole di modularità e quindi più rigido, può portare ad una maggiore efficienza, visto che il codice del kernel è collegato staticamente e quindi l'accesso alle varie funzionalità è immediato. L'approccio del kernel Linux, in qualche modo, cerca di sfruttare i vantaggi di entrambe le "scuole": esso, infatti, è sostanzialmente un kernel monolitico (in Figura 3.7 vediamo che la gran parte dei sotto-sistemi implementati sono statici) con un ampio supporto alla modularizzazione tramite l'utilizzo di *moduli dinamici*, che possono essere aggiunti e tolti in *runtime*. Inoltre però, una volta aggiunto, il codice di ogni modulo è equiparabile al codice collegato staticamente al kernel, eliminando quindi la necessità dello scambio di messaggi e mantenendo l'efficienza: infatti, i moduli possono comunicare tra loro e con le funzionalità principali del kernel semplicemente tramite l'invocazione delle funzioni che implementano tali funzionalità.
I *device driver* fanno appunto parte dei moduli del kernel. I device driver si occupano di fornire uno strato software che nasconde alle applicazioni utente i dettagli dei vari dispositivi hardware, fornendo un'interfaccia composta da funzioni standard (le *system call*) che le applicazioni possono richiamare, dove ognuna di queste funzioni viene poi implementata dal driver per "mappare" la richiesta dell'utente sui comandi specifici per il dispositivo.
Esistono tre tipi fondamentali di dispositivi che i device driver devono gestire, così come vengono descritti in [4], anche se questa distinzione non è rigida:

- *dispositivi a caratteri*: vengono acceduti come uno stream di byte. Per questi dispositivi vengono sempre messe a disposizione system call fondamentali come *open*, *close*, *read* e *write*. Inoltre, essi vengono rappresentati all'interno del sistema come speciali file che si trovano dentro la directory */dev*;

- *dispositivi a blocchi*: nei sistemi Unix, tradizionalmente, questi dispositivi si distinguevano da quelli a caratteri perché essi permettevano soltanto trasferimenti di *blocchi* di dati (solitamente di 512 byte). In Linux, comunque, questa distinzione non è mantenuta, in quanto il kernel Linux permette di accedere ad un dispositivo a blocchi per trasferire un numero qualsiasi di byte, così come è per quelli a caratteri. Inoltre, anche questi dispositivi sono rappresentati tramite file sotto */dev*. In generale, comunque, viene definito dispositivo a blocchi un dispositivo

in grado di ospitare un filesystem. Inoltre, la rappresentazione interna al kernel, così come l'interfaccia tra il kernel e il device driver, è diversa tra i dispositivi a blocchi e quelli a caratteri, per cui è necessario mantenere la distinzione;

- *interfacce di rete*: dispositivi progettati per scambiare pacchetti di dati con altri *host*. Questi non vengono rappresentati all'interno del filesystem, e la comunicazione con essi risulta completamente differente rispetto ai dispositivi a caratteri o a blocchi.

Oltre a questa categorizzazione, esistono comunque dei dispositivi che necessitano di una maggiore stratificazione logica per implementarne il protocollo e, dunque, sono in qualche modo "ortogonali" alle definizioni date: è questo il caso dei dispositivi SCSI o USB (questi ultimi verranno visti nel dettaglio più avanti, cfr. 3.3.5). Per essi esistono nel kernel dei sotto-sistemi ad hoc che gestiscono il protocollo a basso livello, per cui si parla in generale di "dispositivo SCSI" o "dispositivo USB", ma questi poi possono implementare indipendentemente dispositivi a caratteri, a blocchi o interfacce di rete.

3.3.1 I moduli del kernel Linux

Come detto nella sezione precedente (cfr. 3.3) i moduli del kernel riescono a mantenere un'efficienza del tutto paragonabile al codice collegato staticamente, evitando lo scambio di messaggi. Questo meccanismo è possibile grazie al concetto di *simbolo*, dove ogni simbolo rappresenta una funzione o una variabile che viene resa "pubblica" all'interno del kernel. Qui, tre sezioni all'interno del codice del kernel sono usate per mantenere le informazioni sui simboli a cui si può far riferimento all'interno del kernel: la `__kstrtab` contiene l'elenco dei nomi dei simboli disponibili, mentre la `__ksymtab` e la `__ksymtab_gpl` registra gli indirizzi di memoria, rispettivamente, dei normali simboli del kernel e di quelli che vengono esplicitamente detti compatibili con la licenza GPL. Infatti, il kernel Linux, per promuovere lo sviluppo di software libero e scoraggiare l'inserimento di parti proprietarie all'interno del proprio codice, utilizza fondamentalmente due strategie: far sì che parti sostanziali del kernel (come l'algoritmo di gestione della memoria o di scheduling) siano incluse staticamente nel kernel e rendere disponibili alcune funzionalità del kernel soltanto ai moduli che dichiarano di essere compatibili con la licenza GPL.
Oltre alle sezioni descritte per la gestione dei simboli del kernel, ogni modulo (descritto all'interno del kernel da una apposita struttura `module`) contiene l'insieme dei simboli che pubblica usando gli elementi `syms` e `gpl_syms` della

struttura `module`. Così, in fase di linking, tali simboli possono essere pubblicati nel sistema all'interno del file */proc/kallsyms*, per poi essere cancellati da questo file quando il modulo è scollegato.
Quindi, a causa del fatto che i moduli vengono eseguiti in un contesto dove molte risorse vengono naturalmente condivise, la programmazione dei moduli del kernel richiede una serie di considerazioni importanti che non sempre si applicano alli normali software di livello applicativo:

1. mentre di solito le applicazioni svolgono una procedura dall'inizio alla fine, i moduli si "presentano" diversamente: ogni modulo inizia il suo ciclo di vita registrando se stesso (tramite una funzione dichiarata con la parola chiave `__init`) all'interno del sistema, pubblicando così le funzionalità (simboli) che fornisce. Una volta fatto questo, le applicazioni e il kernel possono richiamare in ogni momento queste funzionalità. Quando il modulo viene rimosso dal kernel, le sue funzionalità vengono semplicemente rese non più disponibili (e viene richiamata una funzione di uscita dichiarata come `__exit`);

2. i moduli non possono utilizzare nel loro codice librerie esterne al kernel. In particolare, ad esempio, non è permesso l'uso delle librerie standard;

3. mentre un fallimento da parte di un'applicazione viene gestito in maniera sicura dal kernel, se un modulo del kernel fallisce questo porta sicuramente alla morte del processo corrente, se non al blocco dell'intero sistema;

4. i moduli del kernel lavorano in *kernel space*, dove si possono effettuare operazioni con privilegi superiori rispetto allo *user space* e, inoltre, il kernel non effettua controlli di integrità per le operazioni effettuate dal proprio codice (considerandolo chiaramente affidabile), mentre quanto arriva dallo *user space* è sempre verificato prima di essere utilizzato;

5. **ogni** modulo del kernel va programmato considerando la *concorrenza*. Infatti, esistono diverse fonti di concorrenza all'interno del kernel:

   (a) la presenza di più processi che possono richiedere contemporaneamente le funzionalità dei moduli;

   (b) la possibilità dell'arrivo di *interrupt* che possono intervenire in maniera asincrona imponendo l'esecuzione di un apposito handler, interrompendo così l'esecuzione del codice del driver in un punto arbitrario;

(c) la possibilità che uno stesso modulo possa essere eseguito concorrentemente: questo può avvenire sia per la presenza di architetture SMP ma anche, dal kernel 2.6, a causa del fatto che il kernel è stato reso *preemtible*.

A causa di questo fatto è necessario implementare ogni modulo utilizzando i meccanismi di sincronizzazione (come mutex, *spinlock*, ecc) messi a disposizione dal kernel;

6. i device driver devono fornire *meccanismi* e non *politiche*: mentre i primi implementano una generica astrazione dell'hardware, che appare al livello applicativo come una "scatola nera" su cui è possibile effettuare diverse operazioni, le seconde forniscono le regole e i protocolli con cui utilizzare le varie operazioni permesse sull'hardware. L'unico caso in cui, di solito, vengono implementati all'interno dei device driver aspetti che riguardano le politiche è la *sicurezza*, ad esempio quando l'ordine con cui le operazioni vengono effettuate ha una particolare semantica quando si interagisce con l'hardware. Si pensi al riordinamento che possono subire alcune operazioni a causa del compilatore che cerca di ottimizzare l'efficienza: se si parla di due semplici scritture su una linea seriale, però, questo riordinamento può portare a risultati completamente diversi e non desiderati;

7. mentre le applicazioni, che utilizzano i meccanismi di memoria virtuale implementati nel sistema, possono disporre di un'ampia quantità di spazio nello stack personale, il kernel mette a disposizione soltanto pochissimo spazio di stack (solitamente una singola pagina) che viene condiviso da tutto il kernel space. Per questo è sempre sconsigliabile dichiarare grandi variabili locali nel codice di un modulo, ma, se si necessita di strutture molto grandi, è preferibile utilizzare le funzioni di gestione dinamica della memoria interne al kernel (come la `kmalloc`);

8. in particolare, il kernel non utilizza per l'allocazione di memoria una strategia "heap-like" come le applicazioni in user-space, ma manipola la memoria *fisica* del sistema, che è disponibile soltanto in chunck della dimensione di una pagina. Per questo, la funzione `kmalloc` può ritornare delle quantità di memoria maggiori di quelle richieste (di solito, minimo 32 o 64 byte) e ha un limite di allocazione massimo (di solito intorno ai 128 KB, ma è dipendende dall'architettura e dalla configurazione del kernel), superato il quale è necessario utilizzare altri meccanismi, come lo SLAB (o *lookaside cache*), che permette di allocare degli interi pool di memoria che si possono usare con aree di memoria della stessa

dimensione, o le funzioni come `get_free_page` che permettono di ottenere direttamente intere pagine libere. In ogni caso, il kernel ha la possibilità sia di allocare zone di memoria *virtuale* (tramite la `vmalloc`) sia di ottenere indirizzi virtuali che permettono di accedere porzioni di memoria fisica (tramite la `ioremap`). Infine, per quanto detto finora, l'astrazione usata per catalogare le zone di memoria è diversa tra kernel e user space: infatti, nel kernel esistono 3 zone che vengono considerate:

(a) *normal*, dove l'allocazione viene di solito effettuata;

(b) *DMA-capable*, una zona di memoria preferenziale, dove le periferiche effettuano le operazioni di DMA. In situazioni normali, le pagine libere da allocare vengono ricercare nella normal zone o in questa;

(c) *High memory*, una zona per l'accesso a quantità di memoria maggiori del solito, che però è meno "usabile".

È possibile specificare la zona di memoria nella quale effettuare l'allocazione come flag passato alla funzione `kmalloc`.

9. altra caratteristica molto interessante della programmazione modulare nel kernel Linux è la possibilità di implementare una stratificazione di moduli utilizzando i simboli pubblici: infatti, in questo modo, alcuni moduli possono affidarsi ad altri per la gestione di particolari funzionalità semplicemente richiamando le funzioni appropriate disponibili nella kernel symbol table. Per l'implementazione di questo meccanismo di dipendenza tra moduli, ogni modulo contiene, all'interno della sua `struct module`, un elemento chiamato `modules_which_use_me`, una lista in cui vengono inseriti tutti i moduli che sono dipendenti da esso. All'avvio del sistema, viene invocato il comando `depmod` che scorre tutti i moduli installati nel sistema (ovvero presenti all'interno di */lib/modules*) e ne calcola le dipendenze, salvandole nel file *modules.dep*.
   Grazie alla stratificazione dei moduli, è possibile implementare architetture in cui esistono alcuni moduli più generici che utilizzano dei simboli che poi vengono implementati da moduli di più basso livello che si interfacciano, ad esempio, con un hardware specifico. La Figura 3.8 mostra l'implementazione di tale stratificazione per quanto riguarda il device driver della porta parallela all'interno del kernel 2.6.
   La stratificazione è un'ottima cosa in fase di progettazione, perché permette di mantenere la modularità e la semplicità di ogni modulo. Nel nostro caso, come vedremo più avanti nel dettaglio (cfr. 5 e 6), è stato

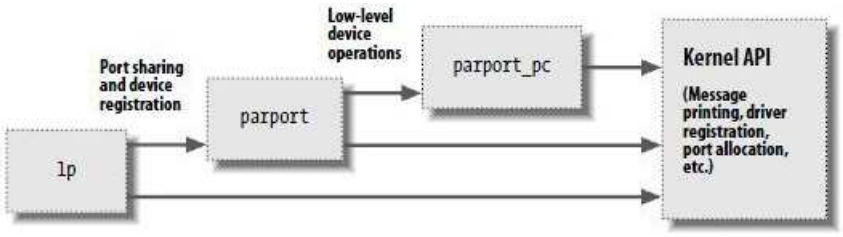

Figura 3.8: Stratificazione dei moduli del driver per la porta parallela nel kernel Linux 2.6. Fonte [4]

interessante implementare tale stratificazione in modo da fornire un'interfaccia generica, come era stato detto in apertura del lavoro (cfr. 1.1), per la gestione dei segnali di voltaggio che arrivano da un qualsiasi strumento musicale analogico; tale interfaccia è poi gestita a basso livello da un altro modulo che si occupa della gestione dell'hardware specifico collegato al sistema.

3.3.2 Scrivere device driver per Linux

Per la seguente trattazione prenderemo in considerazione i dispositivi a caratteri, visto che presentano molte caratteristiche che sono state applicate anche nel caso del Thereminux, oltre a fornire un esempio semplice di utilizzo dei principali concetti relativi alla programmazione di device driver. Come detto, questo tipo di dispositivi è rappresentato come un file sotto */dev*. Ad ognuno di questi file e, dunque, ad ogni dispositivo, sono associati due numeri di identificazione: il *Major Number* e il *Minor Number*. Di questi il primo identifica quale device driver è associato al dispositivo, mentre il secondo identifica esattamente il dispositivo al quale si fa riferimento con quel nodo del filesystem. In questo secondo caso, in realtà, il kernel assume semplicemente che il Minor Number si riferisca al dispositivo implementato dal device driver identificato dal Major Number.
Il nodo del filesystem che rappresenta il dispositivo può essere considerato, per utilizzare una terminologia tipica della programmazione a oggetti, un oggetto, appunto, e questo ha i suoi metodi. Essi sono forniti da una struttura, detta `file_operations`, che viene dichiarata per ogni dispositivo e contiene una serie di puntatori alle operazioni che possono essere effettuate su di esso. Queste operazioni corrispondono alle system call standard richiamabili dal livello applicativo (ad esempio `open`, `close`, `read`, `write`, `ioctl`, ecc) e, infatti, viene eseguita dal kernel una certa operazione definita nella struttura `file_operations` quando arriva una richiesta dal livello applica-

tivo per la system call corrispondente a quella operazione. Ogni operazione dichiarata nella struttura `file_operations`, come detto, è in realtà un puntatore a funzione che può essere fatto corrispondere all'implementazione di quella operazione specifica per il dispositivo, oppure può essere posto a `NULL` se non viene fornita un'implementazione per quella operazione. Ad esempio, all'interno del sistema host del Thereminux viene mostrata la seguente dichiarazione della struttura `file_operations`:

```
static struct file_operations thmx_fops = {
   .owner =    THIS_MODULE,
   .open =     thmx_open,
   .release =  thmx_release,
   .ioctl =    thmx_ioctl,
   .read =     thmx_read,
};
```

Qui si può notare come ogni puntatore a funzione di cui si vuole fornire un'implementazione viene assegnato alla funzione realmente implementata all'interno del driver (si noti come il primo puntatore non punta ad alcuna funzione ma, bensì, ad un identificativo del modulo che "possiede" la struttura dichiarata). Una volta che il driver risulta attivo, dunque, le applicazioni potranno richiamare ad esempio la system call `open` sul file del dispositivo, e questo farà sì che il kernel richiami la funzione assegnata al puntatore `open` dentro `scull_fops`, ovvero la `scull_open`. Questa funzione potrà eseguire ogni inizializzazione di cui necessita la comunicazione con il dispositivo.
Ma come avviene questa "attivazione" del device driver all'interno del sistema?
Dopo aver allocato all'interno del sistema lo spazio (ovvero il Major number) per il device driver, successivamente si procede a registrare il dispositivo, provvedendo alla creazione di un Minor Number, associando al dispositivo la struttura `file_operations` desiderata e, infine, aggiungendo effettivamente il dispositivo al sistema tramite (nel caso di dispositivi a caratteri) la funzione `cdev_add()`. Questo rende il device driver pronto a ricevere richieste in qualsiasi momento.
L'aggiunta del nodo sotto */dev* deve poi essere eseguita separatamente utilizzando il Minor e il Major Number. Questa operazione viene effettuata tramite la system call `mknod` oppure, caso per noi più interessante, tramite `udev`, che permette la creazione dinamica di nodi sotto */dev* a fronte di eventi di *hotplug* (cfr. 3.3.3).
Una volta creato il nodo all'interno del filesystem, dal livello applicativo possono essere eseguite delle richieste tramite le system call standard eseguite

sul file del dispositivo. Non a caso, ad esempio, il puntatore alla system call `open` ha come argomento anche l'inode che si riferisce al nodo del filesystem:

```
int (*open) (struct inode *, struct file *);
```

Tra le altre informazioni, la struttura `inode` contiene i campi `dev_t i_rdev` (che mantiene i numeri di dispositivo) e `struct cdev *i_cdev` (che mantiene la rappresentazione dei dispositivi a caratteri all'interno del kernel).
Altro punto importante, guardiamo i prototipi di alcune operazioni dichiarate all'interno della struttura `file_operations`:

```
int (*open) (struct inode *, struct file *);
ssize_t (*read) (struct file *, char __user *, size_t,
                 loff_t *);
ssize_t (*write) (struct file *, const char __user *,
                  size_t, loff_t *);
...
```

Ci accorgiamo che tutte prendono come argomento (che gli viene passato dal kernel) un puntatore ad una `struct file`. Questa è ben diversa dalla `struct FILE` con cui la *libc* rappresenta i puntatori a file per le applicazioni. La `struct file` identifica all'interno del kernel un *file aperto* e contiene, tra le altre cose, un riferimento alla struttura `file_operations` per quella istanza del dispositivo aperto. Quindi, quando ad esempio una applicazione richiama la system call `read` sul dispositivo, il kernel richiamerà la funzione implementata all'interno del device driver per la `read` e gli passerà un riferimento al file aperto che, a sua volta, contiene un riferimento alla struttura `file_operations` corrente. Questo è utilizzato per implementare una sorta di *method overriding*, in cui un certo metodo è in grado di sostituire l'implementazione dei metodi presenti per il dispositivo in base a certi criteri: ad esempio, all'interno della `open`, il metodo potrebbe controllare i privilegi del processo che ha invocato la system call e modificare i metodi disponibili per il dispositivo di conseguenza, mettendone a disposizione alcuni e nascondendone altri.
Guardando ancora i prototipi mostrati poco sopra, ci accorgiamo anche dell'uso di buffer, per quanto riguarda la `read` e la `write`, dichiarati come `char __user *`. Questo è fatto perché i puntatori dello user space non sono direttamente referenziabili dal kernel: infatti referenziare ciecamente un puntatore dello user space porterebbe ad un evidente problema di sicurezza e, oltre a questo, la gestione della memoria virtuale dello user space potrebbe portare a generare dei *page fault*, cosa che il codice del kernel non ha il permesso di fare. Per questo, esistono delle apposite funzioni, chiamate

`copy_to_user()` e `copy_from_user()` per trasferire in maniera sicura dati tra lo user space e il kernel space. La parola chiava `__user` ci fa capire che si tratta di dati coinvolti in tale trasferimenti e, quindi, occorre prestare particolare attenzione.

3.3.3 Il *Sysfs*

In apertura del lavoro (cfr. 1.1.1 e i requisiti `RHW_F.6` e `RSW_F.1`) è stato specificato che il Thereminux funziona attraverso il meccanismo di *hotplug*, ovvero il software è in grado di gestire dinamicamente il collegamento e la disconnessione fisica dello strumento dal sistema, caricando i moduli necessari e allocando le informazioni di configurazione corrette per lo strumento. Questa funzione, che verrà descritta nel dettaglio in seguito (cfr. 3.3.4), è gestita, nel kernel Linux 2.6, dal *Linux Device Model*, un modello unico per la gestione e il tracciamento di tutti i dispositivi presenti all'interno del sistema, che viene rappresentato tramite il filesystem virtuale chiamato *sysfs*, normalmente posizionato sotto */sys/*.
Per quanto una trattazione approfondita del sysfs non sembra centrale per gli scopi di questo progetto, è importante conoscerne i meccanismi di base per comprendere come il kernel si comporta di fronte a determinate operazioni effettuate dai device driver.
La struttura centrale alla base del Linux Device Model è il *kobject*. Questa struttura mantiene una serie di informazioni importanti, come il numero di riferimenti attivi per un certo dispositivo, i riferimenti ad altri kobject al quale è collegato (per l'implementazione di strutture gerarchiche), la gestione degli eventi di hotplug, ecc. Si può senz'altro dire che ogni "oggetto" che esiste all'interno del sysfs viene rappresentato con un kobject. In particolare, la creazione di un nuovo kobject causa l'allocazione di una nuova directory all'interno del sysfs. I file che queste directory contengono rappresentano gli *attributi* (ad esempio il *vendorId*, il *productId*, ecc) associati al kobject. Moltissime strutture interne del kernel (come, ad esempio, la `struct cdev` di cui si è discusso) hanno associato (tramite il meccanismo del contenimento all'interno della struttura stessa) un kobject che le rappresenta all'interno del device model. Inoltre, quindi, tutte le funzioni di inizializzazione, registrazione, cancellazione, ecc. di strutture che rappresentano dispostivi, bus o device driver eseguono delle operazioni sul sysfs e sui kobject associati a tali strutture. Ad esempio, riprendendo il caso dei driver per dispositivi a caratteri presentato in precedenza (cfr. 3.3.2), ogni driver che voglia implementare un dispositivo a caratteri deve, per prima cosa, inizializzare la struttura `cdev` tramite la funzione `cdev_init(cdev, fops)`, che prende come argomenti la struttura `cdev` e quella `file_operations` con le operazioni implementate su

quel dispositivo. Ebbene, se guardiamo l'implementazione di questa funzione all'interno del sorgente del kernel (file *fs/char_dev.c*), vediamo che risulta così:

```
void cdev_init(struct cdev *cdev, const struct file_operations
               *fops)
{
     memset(cdev, 0, sizeof *cdev);
     INIT_LIST_HEAD(&cdev->list);
     kobject_init(&cdev->kobj, &ktype_cdev_default);
     cdev->ops = fops;
}
```

Come possiamo vedere, dopo alcune operazioni di inizializzazione, la funzione più importante è senza dubbio la `kobject_init()`, che si occupa, naturalmente, di inizializzare il kobject per il dispositivo da aggiungere al sistema. È stata menzionata in precedenza l'organizzazione gerarchica del sysfs. Questa viene ottenuta in due modi. Il primo è il puntatore `parent`, posto all'interno della struttura `kobject`, che punta al kobject che è al livello superiore nella gerarchia. Il secondo modo è l'utilizzo dei *kset*. Questi permettono l'aggregazione di un insieme di kobject dello stesso *tipo*, dove il tipo identifica caratteristiche comuni, come gli attributi di default condivisi e le operazioni che si possono effettuare su tali attributi attraverso il sysfs. Oltre a questo, il device model mette a disposizione, a basso livello, le strutture per la rappresentazione e gestione di bus, device e device driver, che però non verranno approfondite ulteriormente qui.

3.3.4 `udev` e la gestione dell'hotplug

La gestione dell'*hotplug*, come già detto (cfr. 3.3.3), è anch'essa affidata al device model. La trattazione di questa questione richiede qualche attenzione, in quanto la descrizione che ne viene fornita in [4] e [5] si riferisce al kernel 2.6.11, che purtroppo ha subito delle modifiche nelle successive versioni (dalla 2.6.13 [27]) e, quindi, è divenuta purtroppo obsoleta.
Il modello valido fino al kernel 2.6.12, infatti, prevedeva che il kernel generasse, attraverso il device model, un *evento di hotplug* ogniqualvolta avveniva un cambiamento nella configurazione del sistema, ovvero ogni volta che veniva creato o eliminato un kobject. Questo evento veniva catturato, nello user space, da alcuni *script di hotplug*, che potevano eseguire diverse operazioni utilizzando variabili di ambiente che venivano impostate dal kernel come parte dell'evento di hotplug. Il programma *udev* si limitava ad utilizzare le

informazioni pubblicate nel sysfs per creare i nodi dei dispositivi nel filesystem.
I sistemi attuali implementano un modello interamente basato sul manager dei dispositivi *udev* [27] [28]. Udev è un applicativo formato da un demone (*udevd*) che rimane in ascolto (su un socket di tipo `PF_NETLINK` e sulla porta `NETLINK_KOBJECT_UEVENT`) di particolari eventi generati dal kernel che vengono chiamati *uevent*. Il kernel genera un uevent impostando variabili di ambiente e quant'altro possa far reagire udev nel modo corretto. Questa "reazione" da parte di udev avviene attraverso una serie di *regole*, che vengono scritte all'interno di file statici (posti, nel sistema di chi scrive, nella directory */etc/udev/rules.d*). Queste regole permettono di individuare le caratteristiche dell'evento avvenuto ed eseguire le operazioni desiderate, come aggiungere un nodo sotto */dev*, eseguire script, caricare moduli nel kernel, ecc. Se, ad esempio, si volesse eseguire uno script dopo il collegamento di un dispositivo USB, una regola potrebbe essere la seguente:

```
SUBSYSTEM=="usb", ACTION=="add",
ATTRS{serial}=="L72010011070626380", RUN+="/usr/sbin/a_script"
```

Questa regola viene portata a termine quando il demone di udev riceve un evento in cui è stato aggiunto (`ACTION==“add”`) un dispositivo del sottosistema USB (`SUBSYSTEM==“usb”`) di cui viene fatto il matching con il numero seriale, che è presentato come attributo nel sysfs. Se l'evento si riferisce al dispositivo giusto, viene eseguito un certo script.
Per quanto riguarda il sottosistema USB, in particolare, la registrazione di un nuovo dispositivo da parte del *core* USB viene accompagnata dall'invio di appositi *uevent* per pubblicare, come variabili di ambiente, i dati relativi al dispositivo (cfr. file *drivers/usb/core/message.c* nel sorgente del kernel):

```
if (add_uevent_var(env, "INTERFACE=%d/%d/%d",
    alt->desc.bInterfaceClass,
    alt->desc.bInterfaceSubClass,
    alt->desc.bInterfaceProtocol))
  return -ENOMEM;

if (add_uevent_var(env,
    "MODALIAS=usb:v%04Xp%04Xd%04Xdc%02Xdsc%02Xdp%02Xic%02Xisc
                  %02Xip%02X",
     le16_to_cpu(usb_dev->descriptor.idVendor),
     le16_to_cpu(usb_dev->descriptor.idProduct),
     le16_to_cpu(usb_dev->descriptor.bcdDevice),
     usb_dev->descriptor.bDeviceClass,
```

```
    usb_dev->descriptor.bDeviceSubClass,
    usb_dev->descriptor.bDeviceProtocol,
    alt->desc.bInterfaceClass,
    alt->desc.bInterfaceSubClass,
    alt->desc.bInterfaceProtocol))
  return -ENOMEM;
```

Questo evento potrà essere catturato da un'apposita regola di udev che, magari, può essere utilizzata per caricare dinamicamente tutti i moduli che hanno dichiarato di poter gestire quel dispositivo (pubblicando, ad esempio, il numero seriale, il vendorId, ecc del dispositivo al quale fanno riferimento). La regola potrebbe apparire come la seguente [26]:

```
ENV{MODALIAS}=="?*",
RUN+="/sbin/modprobe --use-blacklist $env{MODALIAS}"
```

Questa regola verrà portata a termine per tutti gli eventi che hanno impostato la variabile di ambiente `MODALIAS` ed eseguirà il caricamento dei moduli che possono gestire i dati conservati in questa variabile. Per ulteriori dettagli sul sottosistema USB rispetto allo sviluppo di device driver USB, si veda il paragrafo 3.3.5.

3.3.5 I device driver e il sottosistema USB

I driver USB sono dipendenti, naturalmente, dall'apposito sottosistema all'interno del kernel. Il sottosistema USB si occupa di gestire, come descritto accuratamente nella documentazione ufficiale [2], un'interfaccia di comunicazione tra i dispositivi USB e i device driver, in modo tale che questi ultimi possano "ragionare" in termini di *endpoint* e *interfacce* (cfr. 3.2), ma senza gestire i livelli inferiori del protocollo. Il sottosistema USB fornisce automaticamente un Major Number a tutti i device driver USB, che si devono preoccupare di allocare un proprio Minor Number richiamando la funzione `usb_register_dev()`.
Inoltre, ogni device driver deve implementare due funzioni fondamentali: la `probe()` e la `disconnect()`. La prima si occupa delle inizializzazioni specifiche che il driver fa con il dispositivo che gestisce, mentre la seconda si occupa di effettuare la "pulizia" quando un dispositivo viene scollegato dal sistema.
Ma andiamo con ordine. Supponiamo, come nel caso di nostro interesse, di dover collegare un nuovo dispositivo USB per il quale si dispone di un opportuno device driver, che vogliamo caricare dinamicamente. Il device driver deve dichiarare una tabella con gli identificativi del dispositivo che può servire. Questa dichiarazione, nel driver lato host del Thereminux, appare così:

```
static struct usb_device_id thmx_table [] = {
{ USB_DEVICE(USB_THMX_VENDOR_ID, USB_THMX_PRODUCT_ID) },
{ } /* Terminating entry */
};
MODULE_DEVICE_TABLE (usb, thmx_table);
```

Una volta che il device driver è installato nel sistema, ovvero inserito in una directory standard per i moduli del sistema (nel caso di chi scrive */lib/modules*) e registrato con le sue eventuali dipendenze tramite il comando `depmod`, colleghiamo il dispositivo. Questo evento viene rilevato dal kernel che incarica il *core* USB di gestirlo [26]. Il *core* USB utilizza un *driver USB generico* (implementato in *drivers/usb/core/generic.c*) per effettuare delle operazioni comuni a tutti i dispositivi USB, tra cui comunicare con il dispositivo richiedendogli le informazioni sulle interfacce, sulle configurazioni e sui vari descrittori che lo identificano (cfr. 3.2) e, così, allocare una nuova interfaccia USB (ovvero una `struct usb_interface`) con le caratteristiche inviategli da esso. Il driver generico, a questo punto, esegue una scansione di tutti i driver USB già caricati nel sistema confrontando gli indentificativi invatigli dal dispositivo con quelli che ogni driver dichiara (attraverso la tabella degli identificativi) di poter gestire. Quando trova un driver, tra quelli già caricati nel kernel, che dichiara di poter gestire il dispositivo, il driver generico richiama la funzione `probe()` implementata da quel driver. Infine, genera un *uevent* in cui pubblica le informazioni del dispositivo come variabili di ambiente e questo fa sì che, se il driver che può gestire il dispositivo non era stato ancora caricato, il caricamento possa avvenire: infatti, l'uevent generato risveglia `udev` che, attraverso una opportuna regola (cfr. 3.3.4), richiama il comando `modprobe`, il quale esegue una scansione dei moduli disponibili e, se ne trova uno adatto al dispositivo, lo carica rispettando le sue dipendenze. In questo modo il device driver del Thereminux, inizialmente non presente, viene caricato. Gli eventi generati da `udev` possono essere monitorati, da shell, tramite il tool `udevadm monitor` che, a fronte della connessione di un nuovo dispositivo usb, sul sistema di chi scrive risponde come segue:

```
UEVENT[1273613041.732444] add
/devices/pci0000:00/0000:00:1d.7/usb2/2-1 (usb)
UDEV  [1273613041.735635] add
/devices/pci0000:00/0000:00:1d.7/usb2/2-1 (usb)
UEVENT[1273613041.735677] add
/devices/pci0000:00/0000:00:1d.7/usb2/2-1/2-1:1.0 (usb)
UEVENT[1273613041.735698] add
/class/scsi_host/host28 (scsi_host)
UEVENT[1273613041.735717] add
```

```
/devices/pci0000:00/0000:00:1d.7/usb2/2-1/2-1:1.0/usb_endpoint/
usbdev2.12_ep81 (usb_endpoint)
UEVENT[1273613041.735736] add
/devices/pci0000:00/0000:00:1d.7/usb2/2-1/2-1:1.0/usb_endpoint/
usbdev2.12_ep01 (usb_endpoint)
UDEV  [1273613041.737999] add
/devices/pci0000:00/0000:00:1d.7/usb2/2-1/usb_endpoint/
usbdev2.12_ep00 (usb_endpoint)
UDEV  [1273613041.821581] add
/devices/pci0000:00/0000:00:1d.7/usb2/2-1/2-1:1.0 (usb)
UDEV  [1273613041.822351] add
/class/scsi_host/host28 (scsi_host)
UDEV  [1273613041.823172] add
/devices/pci0000:00/0000:00:1d.7/usb2/2-1/2-1:1.0/
usb_endpoint/usbdev2.12_ep81 (usb_endpoint)
UDEV  [1273613041.824074] add
/devices/pci0000:00/0000:00:1d.7/usb2/2-1/2-1:1.0/
usb_endpoint/usbdev2.12_ep01 (usb_endpoint)
UEVENT[1273613046.733427] add
/devices/pci0000:00/0000:00:1d.7/usb2/2-1/2-1:1.0/
host28/target28:0:0/28:0:0:0 (scsi)
UEVENT[1273613046.733482] add
/class/scsi_disk/28:0:0:0 (scsi_disk)
UEVENT[1273613046.741003] add      /block/sdd (block)
UEVENT[1273613046.741041] add      /block/sdd/sdd1 (block)
UEVENT[1273613046.741060] add
/class/scsi_device/28:0:0:0 (scsi_device)
UEVENT[1273613046.741079] add
/class/scsi_generic/sg4 (scsi_generic)
...
```

Come si può notare, vengono inviati eventi man mano che il dispositivo viene registrato all'interno del sysfs.
L'inserimento del nuovo device driver, a seguito della regola di udev eseguita, causa una nuova scansione dei driver da parte del *core* USB che, quando ne trova uno che può gestire il dispositivo (magari il driver appena collegato), chiama la funzione `probe()` del driver. In questo modo, il driver può effettuare le inizializzazioni vere e proprie del dispositivo, estraendo dalla `struct usb_interface` che gli viene passata come argomento le specifiche degli endpoint per poterli allocare. Ogni endpoint viene rapprentato come una `struct usb_host_endpoint`, che a sua volta contiene una `struct`

`usb_endpoint_descriptor` con le informazioni vere e proprie sull'endpoint. Quindi, dal punto di vista del device driver USB, possiamo supporre che esso venga caricato nel sistema dinamicamente, tramite una regola di udev, quando il dispositivo viene collegato, oppure manualmente in qualsiasi altro momento. In ogni caso, nella sua funzione di `__init`, il driver deve registrare le proprie informazioni più importanti (nome del driver, tabella degli identificatori, puntatori alle funzioni `probe()` e `disconnect()`) tramite la funzione `usb_register()`. Fatto questo, quando il dispositivo viene collegato e il *core* USB identifica il device driver come adatto a gestirlo, viene richiamata la funzione `probe()` del driver stesso. In questa funzione il driver estrae dall'interfaccia USB che gli viene fornita le informazioni sugli endpoint specifici per effettuare la comunicazione con il dispositivo, per poi richiamare la `usb_register_dev()`, con cui si registra la struttura `file_operations` per questo device driver e si ottiene un Minor Number.
Infine, trattiamo la questione di come la comunicazione effettiva tra device driver e dispositivo USB avviene. Tale comunicazione è effettuata tramite i cosiddetti *USB Request Block* (URB). Un URB permette una comunicazione asincrona tra un device driver e un determinato endpoint all'interno del dispositivo, e viene rappresentata come una `struct urb`. Un URB ha il seguente ciclo di vita:

- viene allocato tramite la funzione `usb_alloc_urb()`;
- viene preparato per il suo utilizzo vero e proprio inserendo nella struttura `urb` tutti i dati che si riferiscono a quel trasferimento. Per questo esistono delle funzioni *helper* per i trasferimenti di tipo control, bulk o interrupt (rispettivamente, `usb_fill_control_urb()`, `usb_fill_bulk_urb()` e `usb_fill_int_urb()`), mentre per i trasferimenti di tipo isochronous (cfr. 3.2) l'inizializzazione deve essere fatta a mano. Sostanzialmente deve essere specificato, all'interno della struttura `urb`, il dispositivo e l'endpoint verso/da il quale il trasferimento deve avvenire, il *return handler* che viene eseguito al termine dell'urb, il buffer dei dati di trasferimento, più altri parametri dipendenti dal tipo di urb. Ad esempio, per un trasferimento di tipo isochronous, si può specificare il numero di *frame* che possono essere mandati all'interno di un singolo urb e la frequenza di trasferimento;
- viene inviato al *core* USB con la funzione `usb_submit_urb()`. Questo passa il controllo al *core* USB che si preoccupa di effettuare effettivamente il trasferimento;
- al termine, viene richiamata la funzione *handler*;

- ogni urb può essere cancellato in qualsiasi momento. Questo viene fatto tramite due funzioni: la `usb_kill_urb()`, richiamata di solito quando il dispositivo viene disconnesso dal sistema, e la `usb_unlink_urb()`, quando il device driver vuole annullare un certo urb.

In realtà, quando si tratta di invare semplici messaggi di tipo *bulk* o *control* è possibile evitare la procedura di gestione di urb, ma esistono delle apposite funzioni sincrone, chiamate `usb_bulk_msg()` e `usb_control_msg()`, che permettono tale scambio di dati specificando molti meno dettagli.
Infine, altre due funzioni che possono essere molto utili per far sì che il dispositivo comunichi dati importanti, come particolari descrittori o stringhe non comunicati nella fase di setup, esistono le apposite funzioni `usb_get_descriptor()` e `usb_get_string()`.

3.4 Lo standard MIDI

Il protocollo **MIDI** (Musical Instrument Digital Interface) è uno standard commerciale dal 1983, che implementa un linguaggio completo di descrizione musicale in forma digitale. La Midi Manufacturers Association (MMA [29]), infatti, ha definito in maniera chiara ed esplicita ogni livello dello standard (hardware, protocollo di comunicazione, messaggi ammessi) in modo da creare un modello condiviso da tutti per la comunicazione, soprattutto in vista della necessità, evidente dal punto di vista musicale, di interconnettere macchine di tipo e costruttore diversi.
Per la completa realizzazione della porta di MIDI-out nello strumento (cfr. requisito `RHW_F.4`) sarebbe stato previsto lo sviluppo di una interfaccia hardware per la conversione dei segnali elettrici generati dalla porta seriale (RS-232) in segnali TTL compatibili con il MIDI (come descritto in 4.1). In ogni caso, come già detto in apertura della relazione (cfr. 1), in questa fase dello sviluppo non sono stati implementati dispositivi hardware, ma è comunque stato realizzato tutto il sistema software che permette l'uso del MIDI-out. Per questo motivo, in questa sezione verranno trattati gli aspetti relativi alla gestione del protocollo e dei messaggi MIDI (cfr. requisito `RSW_F.4`), ma non il livello di descrizione hardware.

3.4.1 Il protocollo software e i messaggi MIDI

Il protocollo MIDI si basa, lato software, sullo scambio di *messaggi* codificati in stringhe di 8 bit [30]. Ogni messaggio è composto da uno o più byte. Il primo, chiamato *Status Byte*, è identificato dal MSB posto a 1, mentre tutti gli altri byte hanno questo bit posto a 0. Per la programmazione in

C, ricordiamo che questi numeri si esprimono in esadecimale, per cui vale la seguente corrispondenza:

```
0 - 0000
1 - 0001
2 - 0010
3 - 0011
4 - 0100
5 - 0101
6 - 0110
7 - 0111
8 - 1000
9 - 1001
A - 1010
B - 1011
C - 1100
D - 1101
E - 1110
F - 1111
```

Dunque un byte si può esprimere con l'uso di due numeri esadecimali e possiamo dedurre che lo Status Byte può andare da `0x80` a `0xFF` (da notare che l'MSB è sempre posto a 1), mentre i successivi eventuali byte del messaggio hanno un range che va da `0x00` a `0x7F` (MSB sempre posto a 0). I bit dello Status Byte (di cui abbiamo trattato soltanto il significato del MSB) possono essere divisi in due gruppi da quattro bit ciascuno. Il primo gruppo (che contiene il MSB e quindi può andare da `0x8` a `0xF`) specifica il *tipo* del messaggio, mentre i rimanenti quattro bit specificano il *canale* sul quale questo messaggio viene inviato. Infatti, il MIDI può inoltrare messaggi su 16 diversi canali logici (da `0x0` a `0xF`). Lo Status Byte, quindi, si compone come segue: 4 bit per il tipo di messaggio e 4 per il canale. In realtà, i messaggi codificabili con i primi quattro bit sono i seguenti (li vedremo in seguito nel dettaglio):

```
0x8 = Note Off
0x9 = Note On
0xA = AfterTouch (ie, key pressure)
0xB = Control Change
0xC = Program (patch) change
0xD = Channel Pressure
0xE = Pitch Wheel
```

Per ognuno di questi è possibile specificare il canale di riferimento con i successivi quattro bit. Gli Status Byte che vanno da `0xF0` a `0xFF`, invece, sono riservati per messaggi che non si riferiscono a nessun canale in particolare. Questi, infatti, sono messaggi che servono a controllare tutti i dispositivi che vengono indirizzati attraverso i vari canali, ad esempio inviando messaggi di sincronizzazione. Vengono divisi in due sotto-categorie: i messaggi *System Common* (da `0xF0` a `0XF7`) e i messaggi *System Realtime* (da `0xF8` a `0xFF`). Le tabelle 3.1 e 3.2 forniscono un quadro riassuntivo di tutti i principali messaggi MIDI con la loro descrizione di base, il range di valori esadecimali che il loro Status Byte può assumere e la presenza o meno di eventuali byte di dati che seguono allo status byte.
In generale, ad esempio, supponiamo di voler suonare una certa nota, diciamo il Do centrale del pianoforte, tramite una tastiera. Quando viene premuto il tasto relativo il dispositivo invia una sequenza di byte simile a quella seguente:

```
0x90 0x3C 0x7F
```

Questo invia un messaggio di Note On (`0x90`, cfr. Tabella 3.1) specificando la Nota numero `0x3C` (cfr. ad esempio [31] per una tabella riassuntiva dei valori delle note nel MIDI) e la *velocity* `0x7F`, ovvero la rapidità di esecuzione della nota (nel MIDI questa viene presa come parametro che definisce l'intensità della nota stessa). Nel momento in cui la nota finisce perché il tasto della tastiera viene rilasciato viene inviato il messaggio seguente:

```
0x90 0x3C 0x00
```

ovvero un messaggio di Note On con velocity uguale a 0 (in alternativa si poteva mandare anche un messaggio di Note Off).
Quando vengono inviati diversi messaggi relativi allo stesso Status Byte, il protocollo MIDI permette di specificare questo Status Byte soltanto la prima volta. In questo modo, ad esempio, lo stream appena esaminato può essere inviato nel modo seguente:

```
0x90 0x3C 0x7F
0x3C 0x00
```

Questo ha esattamente lo stesso effetto che ripetere lo Status Byte, migliorando però l'efficienza della comunicazione.

Messaggi MIDI relativi ad un numero di canale			
Nome	**Descrizione**	**Valore esadecimale**	**Byte di dati**
Note Off	Indica che una certa nota deve essere rilasciata	da `0x80` a `0x8F`	2: il primo specifica il numero di nota (0-127) e il secondo la *velocity* (0-127)
Note On	Indica che una certa nota deve essere suonata. Ogni messaggio di Note On *deve* essere seguito, prima o poi, dal relativo messaggio di Note Off o da un messaggio di Note On con velocity=0	da `0x90` a `0x9F`	2: il primo specifica il numero di nota (0-127) e il secondo la *velocity* (0-127). Una velocity=0 può essere usata per implementare il Note Off
Aftertouch	Nei dispositivi che hanno un sensore della pressione esercitata quando si tiene premuta una nota, questo messaggio permette di controllare questo parametro	da `0xA0` a `0xAF`	2: numero di nota (0-127) e *pressure amount* (0-127)

Controller	Il MIDI mette a disposizione una serie di controller che possono mandare messaggi diversi rispetto al suonare una nota o rilasciarla. Questi vengono gestiti con questi comandi	da`0xB0` a `0xBF`	2: numero del controller (0-127) e valore per quel controller (0-127)
Program Change	Permette di cambiare un determinato *programma* (una patch, uno strumento, o altro, a seconda di come viene interpretato dal dispositivo)	da`0xC0` a `0xCF`	1: numero del programma da impostare (0-127)
Channel Pressure	Permette di controllare la pressione media dei tasti su una tastiera (diverso dall'Aftertouch in cui il controllo di pressione è per la singola nota)	da`0xD0` a `0xDF`	1: valore di pressione (0-127)
Pitch Wheel	Permette di alzare o abbassare il pitch (frequenza) di una nota dell'ordine di *centesimi di nota* (frazioni di mezzo tono)	da`0xE0` a `0xEF`	2: I due byte vengono "fusi" insieme per formare un'unica parola di 14 bit che definisce lo spostamento di frequenza della nota. Sia del primo che del secondo byte vengono considerati i bit da 0 a 6, ponendo i bit da 0 a 6 del primo byte come bit 0-6 della stringa, e i bit 0-6 del secondo byte come bit 7-13 della stringa

Tabella 3.1: Messaggi MIDI relativi ad un numero di canale

Messaggi MIDI comuni a tutti i canali			
Nome	**Descrizione**	**Valore esadecimale**	**Byte di dati**
System Exclusive (SysEx)	Messaggi usati per inviare ad un dispositivo una gran quantità di dati, oppure dei dati specifici per un certo produttore	un primo status byte (deve essere `0xF0`), i byte di dati e l'ultimo byte (deve essere `0xF7`)	Qualsiasi numero di byte compresi tra i due status byte del messaggio. Di solito, il secondo byte specifica il *Manufacturer's ID*
MTC Quarter Frame Message	Messaggio inviato da un device master per sincronizzare temporalmente uno slave	`0xF1`	1: il valore di timecode (0-127)
Song Position Pointer	Messaggio inviato da un device master per far sì che lo slave si posizioni ad un determinato momento dell'esecuzione	`0xF2`	2: come nel caso del Pitch Wheel, questi vengono integrati in un'unica stringa da 14 bit che definisce la battuta musicale alla quale l'esecuzione deve spostarsi
Song Select	Messaggio inviato da un device master per far sì che lo slave suoni un certo brano	`0xF3`	1: numero del brano musicale
Tune Request	Il dispositivo che riceve questo messaggio deve eseguire una ricalibrazione dell'intonazione	`0xF6`	Nessuno
MIDI Clock	Messaggio inviato da un device master per mantenere in sincrono uno slave. Affinché questo possa avvenire, questo messaggio viene inviato ad intervalli regolari	`0xF8`	Nessuno

Tick	Messaggio inviato da un device master per mantenere in sincrono uno slave. Affinché questo possa avvenire, questo messaggio viene inviato ad intervalli regolari di 10ms	`0xF9`	Nessuno
MIDI Start	Messaggio inviato da un device master per far sì che lo slave suoni un certo brano dall'inizio (battuta 0)	`0xFA`	Nessuno
MIDI Stop	Messaggio inviato da un device master per far sì che lo slave interrompa l'esecuzione di un certo brano	`0xFC`	Nessuno
MIDI Continue	Messaggio inviato da un device master per far sì che lo slave suoni un certo brano dalla posizione corrente	`0xFB`	Nessuno
Active Sense	Messaggio inviato da un device ogni 300ms quando non c'è altra attività sul bus. Questo per comunicare agli altri dispositivi che il collegamento è integro	`0xFE`	Nessuno
Reset	Il dispositivo che riceve questo messaggio deve reimpostare se stesso ad uno stato di default, di solito lo stesso che si ha all'accensione	`0xFF`	Nessuno

Tabella 3.2: Messaggi MIDI comuni a tutti i canali

3.5 La rappresentazione dei device MIDI in Linux: il device driver di test *minisnd*

Come specificato con il requisito RSW_F.4, il Thereminux, una volta collegato al pc tramite il cavo USB, appare al livello applicativo come un dispositivo MIDI standard. Al fine di studiare come implementare questa funzionalità, è stato sviluppato un piccolo device driver di test, chiamato *minisnd*, che si serve di un dispositivo USB qualsiasi (nel caso specifico, è stata utilizzata una comune pen drive) e lo inizializza in modo tale che software musicali standard per Linux possano riconoscerlo come dispositivo per l'acquisizione di dati MIDI. Non è stata implementata alcuna funzione per la reale lettura dei dati, visto che l'hardware reale era ben diverso da un dispositivo MIDI. Lo sviluppo del driver *minisnd* ha richiesto lo studio di ALSA, *Advanced Linux Sound Architecture* [32] [33] [34]. ALSA fornisce un insieme di driver, una libreria applicativa e diverse utility per la gestione del sistema audio in Linux. Dal punto di vista delle applicazioni, esistono ormai decine di software professionali che seguono lo standard ALSA, tra cui il progetto *Jack* spicca con particolare rilevanza. Questo fornisce un sistema di configurazione e routing per tutti i software ALSA-compliant attivi nel sistema. La Figura 3.9 mostra l'applicativo grafico Jack-Control.

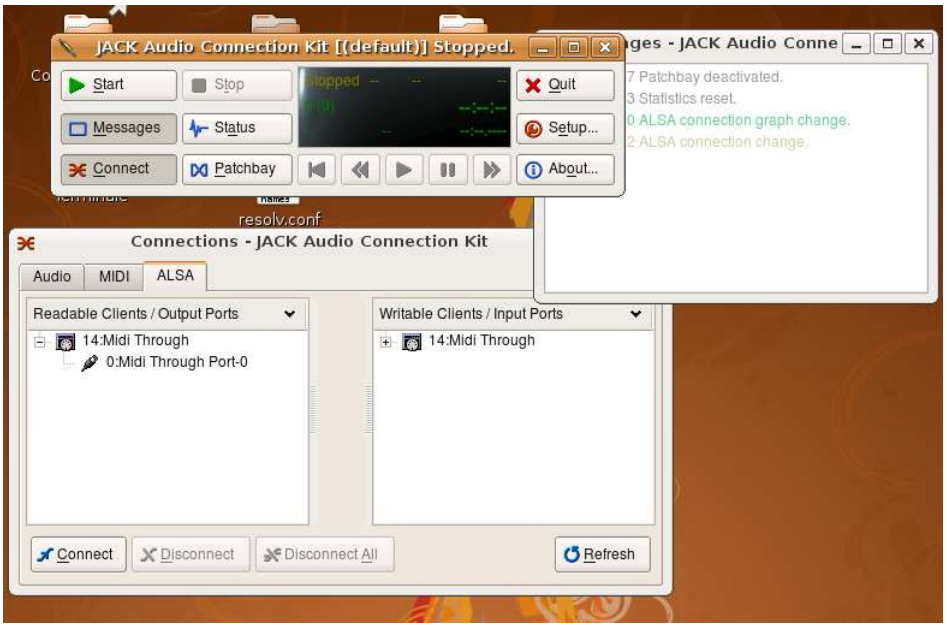

Figura 3.9: Uno screenshot dell'applicazione QjackCtl

Come detto, Jack fornisce un'interfaccia generica per il controllo di tutti i

software e i dispositivi ALSA nel sistema. Come si vede in Figura, mette a disposizione un grafo di routing per redirigere i vari flussi elaborativi da un software all'altro o da un dispositivo all'altro. Ad esempio, è possibile acquisire i dati da una tastiera MIDI, inviarli ad un software per l'elaborazione di questi (ad esempio, il software Qsynth [51] che permette di inserire effetti ed effettuare trasformazioni sul segnale di input) ed inoltrare il segnale di uscita di questo software ad un altro software, che magari registrerà il risultato.
Quello che si vuole scrivere è, dunque, un driver che permetta di mostrare il Thereminux come dispositivo MIDI che viene elencato tra quelli standard presentati da Jack. Fare questo significa inizializzare il dispositivo MIDI non solo tramite la procedura normale di gestione dei disposivi USB discussa in precedenza (cfr. 3.3.5) ma utilizzando in più le funzioni messe a disposizione dall'API ALSA, al fine di "incapsulare" il dispositivo USB dentro un dispositivo MIDI ALSA-compliant. La procedura per lo sviluppo di un device driver ALSA è sostanzialmente la seguente:

1. allocazione di una *sound card*: infatti, ogni dispositivo che è in grado di poter emettere o acquisire eventi sonori è visto, da parte di ALSA, come una scheda sonora, ovvero un dispositivo che contiene l'implementazione delle varie funzionalità (audio, MIDI, ecc). ALSA identifica ogni scheda sonora tramite identificativi del tipo `hw:n`, dove `n` è il numero della scheda sonora;

2. aggiunta dei vari dispositivi alla sound card: come accennato in precedenza, la scheda sonora è soltanto un "contenitore" nel quale vanno successivamente allocati tutti i dispositivi realmente implementati, come quelli per l'acquisizione e/o riproduzione audio/MIDI. Gli identificatori di tali dispositivi seguono da quelli delle sound card: ad esempio, il primo dispositivo della prima scheda è identificato con `hw:0,0` e così via;

3. registrazione della sound card e dei suoi dispositivi: in tutta la fase di inizializzazione, la scheda sonora era vista come non accessibile dal livello applicativo. Con la chiamata alla funzione `snd_card_register()`, successivamente, questa viene attivata. In particolare, vengono creati tutti i collegamenti relativi alla sound card e ai suoi dispositivi sia all'interno del filesystem `/proc` che `/sys`. In quest'ultimo, vengono creati tutti i kobject relativi ai dispositivi presenti nella sound card rispettando la loro gerarchia all'interno del sistema, ovvero mostrando i vari dispositivi sonori, ad esempio, come figli del bus al quale sono collegati. All'interno del filesystem `/proc`, invece, vengono creati dei

collegamenti standard di più facile consultazione. Uno tra tutti, il file */proc/asound/cards* contiene la lista di tutte le sound card presenti nel sistema e, per ognuna, viene mostrata un'apposita directory (ad esempio, */proc/asound/card0* ecc) con i dettagli delle varie schede;

4. infine, la chiamata a `snd_card_free()` libera tutte le risorse precedentemente allocate, compresa la scheda sonora e tutti i dispositivi ad essa associati.

3.5.1 Dettagli implementativi del driver *minisnd*

Come già accennato in precedenza (cfr. 1.1 e 3.3.2) una buona idea per l'implementazione del device driver del Thereminux sembrerebbe essere quella di utilizzare una stratificazione di moduli del kernel, in cui gli strati più ad alto livello dovrebbero fornire funzioni generiche e quelli più "in basso" dovrebbero gestire concretamente la comunicazione con l'hardware. Il device driver *minisnd* è stato già progettato secondo queste considerazioni, in modo da effettuare un primo esperimento sulle modalità implementative di tale stratificazione. La struttura che ne è risultata è mostrata dal diagramma in Figura 3.10.

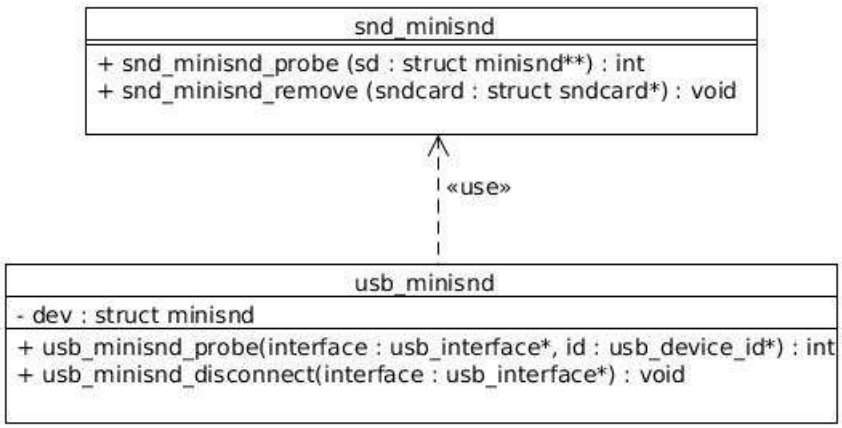
snd_minisnd
+ snd_minisnd_probe (sd : struct minisnd**) : int
+ snd_minisnd_remove (sndcard : struct sndcard*) : void
«use»
usb_minisnd
- dev : struct minisnd
+ usb_minisnd_probe(interface : usb_interface*, id : usb_device_id*) : int
+ usb_minisnd_disconnect(interface : usb_interface*) : void

Figura 3.10: Diagramma dei moduli che costituiscono il driver minisnd

Come si può vedere, sono stati implementati due diversi moduli. Il primo, quello a più alto livello, è `snd_minisnd`, che fornisce una serie di funzioni che servono ad implementare un dispositivo di MIDI input basato su ALSA. I dettagli del dispositivo vero e proprio vengono gestiti dal modulo `usb_minisnd`, che implementa un driver USB generico e richiama le funzioni

messe a disposizione da `snd_minisnd` per la registrazione degli aspetti riguardanti ALSA. La pubblicazione delle funzioni da parte di `snd_minisnd` è effettuata esportando degli opportuni simboli nella Kernel Symbol Table tramite la `EXPORT_SYMBOL`. Infine, tutte le informazioni vengono scambiate tra i due moduli utilizzando un'unica struttura, la `struct minisnd`, che include tutte le informazioni necessarie all'implementazione delle funzionalità. Nel dettaglio, la struttura è definita così nell'header *minisnd.h*:

```
 struct minisnd {
   struct snd_card *card;
   struct snd_rawmidi *midi_device;
   struct usb_data *usbd;
   struct device *dev;
   int devno;
};

struct usb_data {
   struct usb_device *udev;
   unsigned char *bulk_in_buffer;
   size_t bulk_in_size;
   __u8 bulk_in_endpointAddr;
   __u8 bulk_out_endpointAddr;
};
```

Come si vede, la struttura contiene le risorse fondamentali per l'esecuzione del driver, come i puntatori, specifici di ALSA, della sound card (`struct snd_card`) e del dispositivo MIDI ad essa associato (`struct snd_rawmidi`); inoltre, contiene un puntatore ai dati specifici del dispositivo usb, rappresentati con la struttura `usb_data`. In essa si trovano il puntatore alla `struct usb_device` o agli endpoint su cui la comunicazione vera e propria può avvenire. Nel caso specifico gli unici due endpoint considerati sono uno di input e uno di output di tipo *bulk* (cfr. 3.2.2). Ai fini dell'inizializzazione della sound card da parte di ALSA sono richieste le informazioni relative alla `struct device` a cui il dispositivo fa riferimento, quindi è stato inserito, per comodità, anche un puntatore a questa struttura. Infine, viene usato un contatore per il tracciamento dei dispositivi gestiti dal driver: infatti, nel caso delle sound card, ALSA non può gestirne più di otto con lo stesso driver.
In conclusione, alleghiamo il codice relativo alle funzioni principali (la probe e la disconnect) per il modulo `usb_minisnd` (Figura 3.11) e `snd_minisnd` (Figura 3.12).
La funzione `usb_minisnd_probe()` alloca la struttura `minisnd` del driver e la inizializza con i dati specifici del dispositivo. Per fare questo, effettua una

```
 static int __devinit usb_minisnd_probe(struct usb_interface
 *interface, const struct usb_device_id *id)
{
   static int devno;
   static struct minisnd *dev = NULL;
   struct usb_data *usb_dev = NULL;
   struct usb_host_interface *iface_desc;
   struct usb_endpoint_descriptor *endpoint;
   size_t buffer_size;
   int i;
   int retval = -ENOMEM;

   /* allocate memory for our device state and initialize it */
   dev = kmalloc(sizeof(struct minisnd), GFP_KERNEL);
   if (dev == NULL) {
      err("Out of memory");
      goto error;
   }
   memset(dev, 0x00, sizeof(struct minisnd));

   usb_dev = kmalloc(sizeof(struct usb_data), GFP_KERNEL);
   if (usb_dev == NULL) {
      err("Out of memory");
      goto error;
   }
   memset(usb_dev, 0x00, sizeof(struct usb_data));

   dev->usbd = usb_dev;
   dev->usbd->udev = usb_get_dev(interface_to_usbdev(interface));
   dev->dev = &interface->dev;
   dev->devno = devno;

   /* set up the endpoint information */
   /* use only the first bulk-in and bulk-out endpoints */
   iface_desc = interface->cur_altsetting;
   for (i = 0; i < iface_desc->desc.bNumEndpoints; ++i) {
      endpoint = &iface_desc->endpoint[i].desc;

      if (!dev->usbd->bulk_in_endpointAddr &&
         (endpoint->bEndpointAddress & USB_DIR_IN) &&
         ((endpoint->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK)
           == USB_ENDPOINT_XFER_BULK)) {
         printk(KERN_INFO "we found a bulk in endpoint");
         buffer_size = endpoint->wMaxPacketSize;
         dev->usbd->bulk_in_size = buffer_size;
         dev->usbd->bulk_in_endpointAddr = endpoint->bEndpointAddress;
         dev->usbd->bulk_in_buffer = kmalloc(buffer_size, GFP_KERNEL);
```

```
         if (!dev->usbd->bulk_in_buffer) {
            err("Could not allocate bulk_in_buffer");
            goto error;
         }
      }

      if (!dev->usbd->bulk_out_endpointAddr &&
          !(endpoint->bEndpointAddress & USB_DIR_IN) &&
          ((endpoint->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK)
          == USB_ENDPOINT_XFER_BULK)) {
         printk(KERN_INFO "we found a bulk out endpoint");
         dev->usbd->bulk_out_endpointAddr = endpoint->bEndpointAddress;
      }
   }
   if (!(dev->usbd->bulk_in_endpointAddr &&
       dev->usbd->bulk_out_endpointAddr)) {
      err("Could not find both bulk-in and bulk-out endpoints");
      goto error;
   }

   retval = snd_minisnd_probe(&dev);
   if (retval < 0) {
      err("Could not initialize audio");
      goto error;
   }

   /* save our data pointer in this interface device */
   usb_set_intfdata(interface, dev);

   /* we can register the device now, as it is ready */
   retval = usb_register_dev(interface, &usb_minisnd_class);
   if (retval) {
      /* something prevented us from registering this driver */
      err("Not able to get a minor for this device");
      usb_set_intfdata(interface, NULL);
      goto error;
   }

   /* let the user know what node this device is now attached to */
   printk(KERN_INFO "USB MINISND device now attached to
          USBminisnd-%d",interface->minor);
   return 0;

   error:
      if (dev)
         kfree(dev);
      return retval;
}
```

```
static void __devexit usb_minisnd_disconnect(struct usb_interface
*interface)
{
   struct minisnd *dev;
   int minor = interface->minor;

   /* prevent usb_minisnd_open() from racing usb_minisnd_disconnect() */
   lock_kernel();

   dev = usb_get_intfdata(interface);
   snd_minisnd_remove(dev->card);
   usb_set_intfdata(interface, NULL);

   /* give back our minor */
   usb_deregister_dev(interface, &usb_minisnd_class);

   unlock_kernel();

   printk(KERN_INFO "USB MiniSound #%d now disconnected", minor);
}
```

Figura 3.11: Sorgente delle funzioni principali del modulo `usb_minisnd`

scansione degli endpoint presenti nel dispositivo e carica soltanto quelli voluti. Infine, richiama la funzione `snd_minisnd_probe()` per l'inizializzazione della parte relativa ad ALSA e, per concludere, registra il dispositivo sul bus USB tramite la `usb_register_dev()`. La struttura con i dati salvati del dispositivo vengono salvati come dati generici della `interface` USB, in modo da poter essere recuperata in seguito.
La `snd_minisnd_probe()` esegue l'inizializzazione del dispositivo come interfaccia MIDI. Per fare ciò, prima di tutto, verifica la disponibilità di slot per questo dispositivo (considerando il limite `SNDRV_CARDS`, posto a otto in *include/sound/core.h*). In seguito, alloca una nuova sound card tramite la funzione `snd_card_create()`. Il parametro `SNDRV_DEFAULT_IDX1`, che vale -1 come definito nel file *sound/core/init.c* all'interno dei sorgenti del kernel, indica ad ALSA di allocare la sound card in un qualsiasi slot libero. Successivamente, dopo aver specificato alcune informazioni riguardanti la sound card, come il driver che la gestisce o le stringhe di descrizione, si associa la sound card al dispositivo vero e proprio a cui fa riferimento tramite la

`snd_card_set_dev()`. Successivamente, si inizia con l'allocare i dispositivi contenuti nella sound card. Il primo rappresenta la sound card stessa, ed infatti è specificato come `SNDRV_DEV_LOWLEVEL`, ovvero un sound driver di basso livello. In seguito, viene allocato il dispositivo MIDI tramite la `snd_rawmidi_new()` a cui vengono passati come argomenti la sound card, una stringa identificativa, il numero del dispositivo, il numero di canali di output e quelli di input e, infine, un puntatore alla struttura `snd_rawmidi` da allocare. Nel nostro caso, questo driver crea soltanto un dispositivo di MIDI input, quindi i parametri sono coerenti con questa scelta. Anche al dispositivo MIDI creato vengono associate le funzioni per la gestione degli eventi e i vari descrittori per la sua identificazione. Infine, la sound card viene registrata presso ALSA tramite la funzione `snd_card_register()`.
La disconnessione dello strumento viene gestita dalla funzione `usb_minisnd_disconnect()` che richiama la `snd_minisnd_remove()`, la quale non fa altro che deallocare la sound card tramite la `snd_card_free()`. Infine, il dispositivo viene deregistrato presso il bus USB tramite la funzione `usb_deregister_dev()`.

```
int snd_minisnd_probe(struct minisnd **sd)
{
   int err;
   int devno;
   struct snd_rawmidi *mididev = NULL;
   struct snd_rawmidi_ops midi_ops;
   struct snd_card *card = NULL;
   char id_card_string[25];
   char id_midi_string[25];
   struct file_operations *fops;
   static struct snd_device_ops ops = {
      .dev_free = minisnd_chip_free,
   };

   if (!(*sd) || !((*sd)->usbd) || !((*sd)->dev)) {
      return -EINVAL;
   }

   devno = (*sd)->devno;

   if (devno >= SNDRV_CARDS)
      return -ENODEV;
   if (!enable[devno]) {
      devno++;
      return -ENOENT;
   }

   snprintf(id_card_string, sizeof(id_card_string),
    "minisnd_card%d", devno);
   snprintf(id_midi_string, sizeof(id_midi_string),
            "minisnd_rawmidi%d", devno);

   err = snd_card_create(SNDRV_DEFAULT_IDX1, id_card_string,
                         THIS_MODULE, devno, &card);
   if (err < 0)
      return -ENOMEM;

   strcpy(card->driver, "minisnd");
   strcpy(card->shortname, id_card_string);
   sprintf(card->longname, "%s", card->shortname);

   snd_card_set_dev(card, (*sd)->dev);

   snd_device_new(card, SNDRV_DEV_LOWLEVEL, *sd, &ops);
   if (err < 0) {
      snd_card_free(card);
      return err;
   }
```

```
   /* MIDI DEVICE */
   err = snd_rawmidi_new(card, id_midi_string, devno, 0, 1,
                         &mididev);
   if (err < 0) {
      snd_card_free(card);
      return err;
   }

   midi_ops.open = my_rawmidi_open;
   midi_ops.close = my_rawmidi_close;

   snd_rawmidi_set_ops(mididev, SNDRV_RAWMIDI_STREAM_INPUT,
                       &midi_ops);
   mididev->private_data = *sd;
   mididev->info_flags = SNDRV_RAWMIDI_INFO_INPUT;
   strcpy(mididev->name, "minisnd_midi_in");

   err = snd_card_register(card);

   if (err < 0) {
      snd_card_free(card);
      return err;
   }

   (*sd)->card = card;
   (*sd)->midi_device = mididev;

   return 0;
}
EXPORT_SYMBOL(snd_minisnd_probe);

void snd_minisnd_remove(struct snd_card *sndcard)
{
   if (! sndcard) {
      printk(KERN_WARNING "snd_minisnd_remove: NULL sndcard!");
      return;
   }
   snd_card_free(sndcard);
}
EXPORT_SYMBOL(snd_minisnd_remove);
```

Figura 3.12: Sorgente delle funzioni principali del modulo `snd_minisnd`

3.5.2 Esecuzione del driver e risultati

Perché il codice scritto utilizzando l'API di ALSA possa essere compilato, per prima cosa, occorre ricompilare il kernel abilitando il supporto ad ALSA, che in alcune versioni del kernel non c'è di default. In particolare, chi scrive ha lavorato con un kernel 2.6.32, sul quale è stato necessario abilitare le seguenti configurazioni:

```
Device Drivers -->
  <M> Sound card support -->
      <M> Advanced Linux Sound Architecture -->
          <M> Sequencer Support
          <M> OSS Mixer API
          <M> OSS PCM (Digital Audio) API
          <*> OSS Sequencer API
          <*> Support old ALSA API
          <*> Verbose procfs contents
          <*> USB sound devices
```

Successivamente, dopo la compilazione dei due moduli, è stato sufficiente inserirli tra i moduli del sistema (in una sotto-directory di */lib/modules*) e, al fine di ricostruire le dipendenze tra `usb_minisnd` e `snd_minisnd`, richiamare il comando `depmod`.
Fatto questo, i moduli sono stati caricati tramite *modprobe*. In particolare, grazie alla dipendenza tra i due, il comando `modprobe usb_minisnd` ha fatto sì che venisse caricato prima `snd_minisnd` e dopo `usb_minisnd`. A quel punto, la nuova scheda audio è stata perfettamente riconosciuta da ALSA. Infatti, il procfs è stato modificato:

```
daniele@danieledevel-laptop:~$ ls /proc/asound/
card0  cards  devices  minisndcard0  modules  oss  seq  timers
version
```

Come possiamo vedere, appare sia la nuova `card0` che il dispositivo `minisndcard0`, che altro non è che un link simbolico alla directory `card0`:

```
 daniele@danieledevel-laptop:~$ ls -l /proc/asound/
totale 0
dr-xr-xr-x 2 root root 0 2010-06-30 19:11 card0
-r--r--r-- 1 root root 0 2010-06-30 19:11 cards
-r--r--r-- 1 root root 0 2010-06-30 19:11 devices
lrwxrwxrwx 1 root root 5 2010-06-30 19:11 minisndcard0 -> card0
-r--r--r-- 1 root root 0 2010-06-30 19:11 modules
```

```
dr-xr-xr-x 2 root root 0 2010-06-30 19:11 oss
dr-xr-xr-x 2 root root 0 2010-06-30 19:11 seq
-r--r--r-- 1 root root 0 2010-06-30 19:11 timers
-r--r--r-- 1 root root 0 2010-06-30 19:11 version
```

Inoltre, la nuova sound card viene elencata anche nel file */proc/asound/cards*:

```
daniele@danieledevel-laptop:~$ cat /proc/asound/cards
0 [minisndcard0   ]: minisnd - minisnd_card0
                     minisnd_card0
```

Infine, è possibile controllare l'avvenuto riconoscimento da parte di Jack. Avviando il setup di Jack-Control, possiamo vedere la sound card `minisnd_card0` come selezionabile tra le sound card disponibili nel sistema (Figura 3.13).

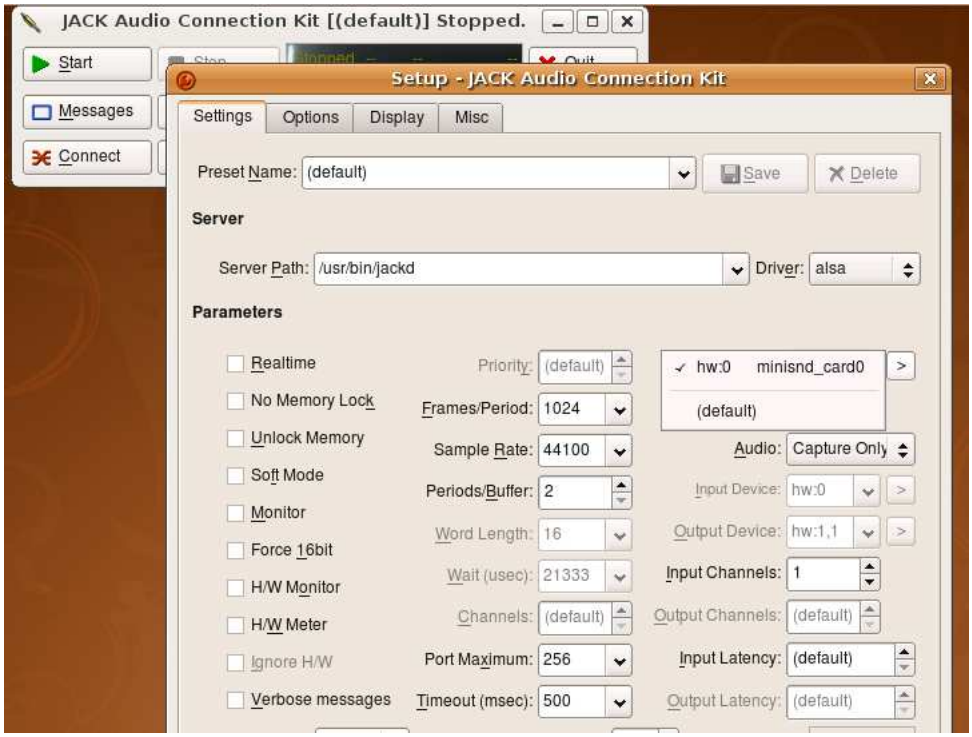

Figura 3.13: Uno screenshot dell'applicazione QjackCtl dopo l'avvenuto riconoscimento della sound card `minisnd_card0`

Oltre a questo, andando nella tabella di routing dei dispositivi e software audio/MIDI, vediamo disponibile il dispositivo di MIDI input che desideravamo (Figura 3.14), sotto il nome `minisnd_midi_in`, che era quello che avevamo impostato all'interno del driver (cfr. Figura 3.12).

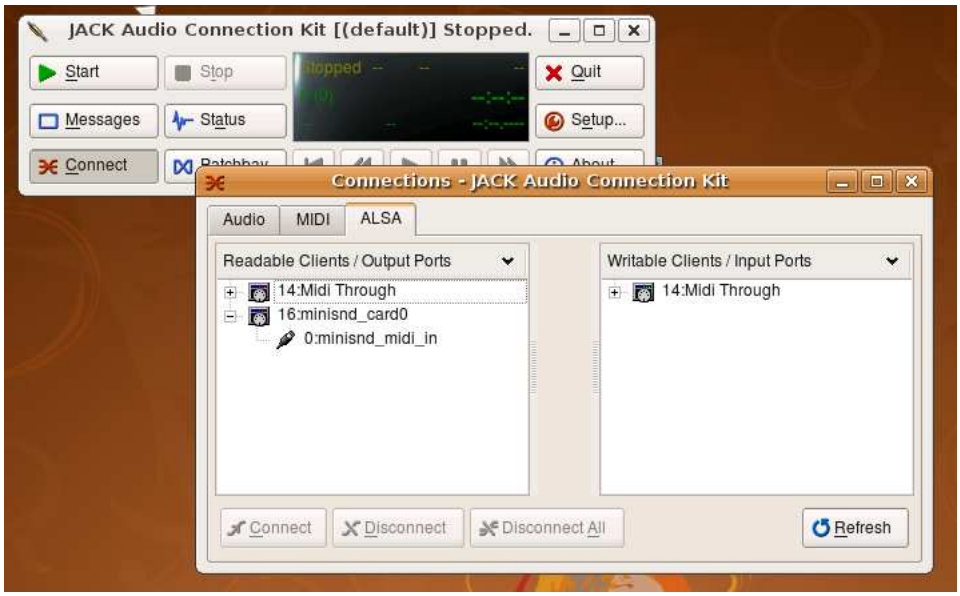

Figura 3.14: Uno screenshot dell'applicazione QjackCtl dopo l'avvenuto riconoscimento del device di MIDI input `minisnd_midi_in`

3.6 La conversione *Voltage-to-MIDI*

Come specificato con il requisito RSW_F.3, il sistema dovrà effettuare una conversione delle stringhe binarie acquisite da input attraverso il bus USB in opportuni messaggi MIDI. Perché questo possa avvenire, sostanzialmente, è necessario riconoscere il *pitch* (nota) del segnale in ingresso e generare la nota MIDI coerente con questo. Naturalmente, però, questa operazione non è banale, visto che l'unica informazione che possiamo ricavare dall' ADC è la lista dei campioni che rappresentano i vari valori dell'*ampiezza* del segnale, mentre il pitch è in relazione con la *frequenza* dell'armonica fondamentale del segnale stesso. Ogni segnale, tra cui quelli sonori, è infatti composto da più componenti sinusoidali dette armoniche o *parziali* e, spesso ma non necessariamente, queste sono in *relazione armonica* con la parziale più bassa, detta fondamentale. La relazione armonica implica che le componenti armoniche superiori alla prima abbiano una frequenza pari ad un multiplo della fondamentale. Effettuare il riconoscimento del pitch relativo ad un certo segnale di ingresso significherà, quindi, estrarre l'informazione relativa all'armonica fondamentale del segnale stesso. Questa operazione è detta *pitch detection* [7] [35].
In generale, gli algoritmi di pitch detection possono ricardere in tre diverse categorie: algoritmi *time-domain*, algoritmi *frequency-domain* e algoritmi

basati su *metodi statistici*.

3.6.1 Algoritmi time-domain

Nel caso di algoritmi time-domain, si cerca di sfruttare la periodicità del segnale nel tempo. Il metodo "zero-crossing" è un tipico esempio di questo approccio, in cui si calcola il periodo del segnale calcolando il tempo che trascorre tra i primi due valori zero del segnale stesso. In questo modo, applicando la semplice relazione matematica tra la frequenza e il periodo ($f = \frac{1}{T}$) si ottiene la frequenza. Il problema fondamentale con questo metodo sta nell'assunzione che all'interno di un segnale lo zero venga attraversato due sole volte per periodo. Ma, nel caso di segnali con alta inarmonicità come quello mostrato in Figura 3.15, questo porta a ricavare un'informazione sbagliata.

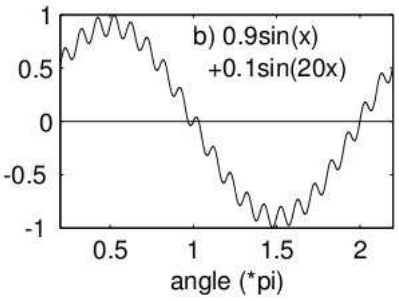

Figura 3.15: Un segnale inarmonico, dove lo zero viene attraversato più di due volte per periodo. Fonte [7]

Un metodo sicuramente più interessante usa la funzione matematica di *autocorrelazione* [7]. Se la correlazione tra due segnali è una misura della loro similarità, l'autocorrelazione è una misura di un segnale con se stesso, ovvero con una sua versione "ritardata" nel tempo. In questo caso, si può chiaramente supporre che i due segnali comparati saranno esattamente identici quando il ritardo temporale è zero (oppure un intero periodo), cioè quando il segnale è in fase con se stesso; in maniera simile, si prevede che i due segnali mostreranno la differenza massima quando il ritardo è di mezzo periodo (ovvero quando il segnale è esattamente fuori fase con se stesso).
Su un segnale limitato temporalmente ad una finestra di grandezza N, la funzione di autocorrelazione è la seguente [7]:

$$R_x(\nu) = \sum_{n=0}^{N-1-\nu} x[n]x[n+\nu]$$

Come si può notare, anche la funzione di autocorrelazione è una funzione periodica e, in particolare, mostra il comportamento detto: quando il segnale è ritardato di mezzo periodo rispetto all'originale (segnale fuori fase con se stesso), la R raggiunge un minimo mentre, quando il segnale con ritardo viene portato ad avere uno shift di un intero periodo (segnale in fase con se stesso), la funzione raggiunge un massimo. Quindi, il primo picco massimo della funzione di autocorrelazione viene solitamente preso come periodo del segnale, da cui dedurre la frequenza portante. Certo, possono comunque nascere dei problemi quando si analizzano segnali complessi come quello in Figura 3.15: infatti, per questo segnale, la funzione di autocorrelazione riconoscerebbe non il periodo principale, ma quello di una delle armoniche superiori del segnale.
Per evitare questo un metodo che sembra avere ottimi risultati sperimentali è lo *YIN* [8]. Seppur basato sull'autocorrelazione, il metodo YIN cerca di minimizzare la differenza tra il segnale e la sua versione ritardata, piuttosto che massimizzare il prodotto tra i due. L'equazione che ne deriva è la seguente:

$$d_t(\tau) = \sum_{j=1}^{W} (x_j - x_{j+\tau})^2$$

In realtà, per ridurre il rischio di avere dei "falsi positivi" nel riconoscimento, l'algoritmo pesa i valori trovati sulla media dei valori precedenti:

$$d'_t(\tau) = \begin{cases} 1, & \tau = 0 \\ \frac{d_t(\tau)}{\frac{1}{\tau}\sum_{j=1}^{\tau} d_t(j)}, & \text{altrimenti} \end{cases}$$

Questo metodo sembra particolarmente interessante per i nostri scopi, dato il fatto che sembra essere adatto alla nostra applicazione e la disponibilità di implementazioni open-source (cfr. 3.7.3).

3.6.2 Algoritmi frequency-domain

Questa classe di algoritmi si basa sulla rappresentazione del segnale campionato tramite le armoniche che lo compongono. Tale rappresentazione è ottenuta tramite metodi standard, come la trasformata di Fourier. I due principali metodi di questa categoria sono l'algoritmo di *maximum likelihood* [10] e il *cepstrum* [9]. Nel primo caso si cerca di trovare uno spettro "abbastanza simile" a quello di input utilizzando molti spettri possibili. Naturalmente, questo algoritmo richiede un certo numero di spettri di confronto per ottenere buoni risultati, e risulta funzionare abbastanza bene nel caso di segnali

con intonazione fissa. Nel secondo caso, invece, viene prodotta la trasformata di Fourier del logaritmo dello spettro di ampiezza del segnale. Infatti, la trasformata di Fourier produce una serie di picchi in corrispondenza delle armoniche del segnale. Utilizzando la scala logaritmica, questi picchi vengono smussati in ampiezza e il segnale di output prodotto diventa un segnale periodico nel dominio della frequenza, in cui la distanza tra i diversi picchi (il periodo) è legata alla frequenza fondamentale del segnale originale. La procedura ora descritta è mostrata in Figura 3.16.

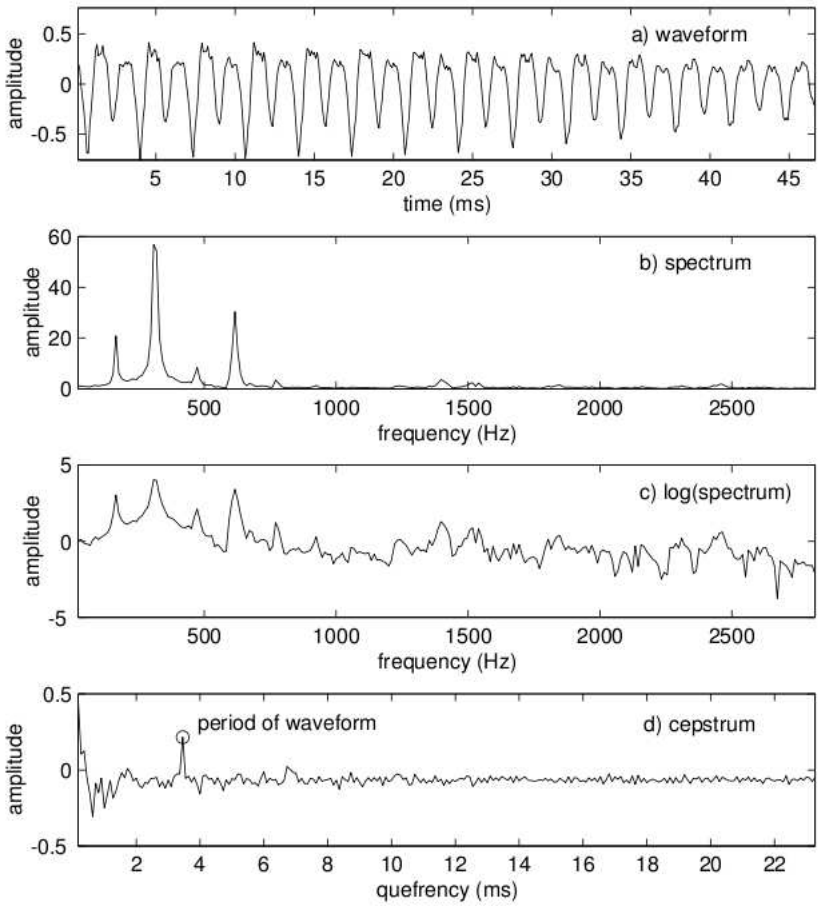

Figura 3.16: La procedura dell'algoritmo *cepstrum*. Fonte [7]

3.6.3 Metodi statistici

Questa classe di algoritmi, piuttosto laboriosa nell'implementazione, cerca di effettuare una classificazione dei diversi frame del segnale tramite algoritmi

di apprendimento automatico. Per questo, tali algoritmi hanno bisogno di un periodo di training la cui accuratezza ne influenza fortemente i risultati finali. La classificazione è effettuata al fine di stabilire se un determinato frame fa parte dell'insieme che rappresenta un certo pitch oppure no. I metodi che sono stati implementati per questa classe utilizzano reti neurali [12] o Hidden Markov Model [11].

3.7 Tecnologie esistenti appropriate per il sistema

Dopo aver studiato, nei paragrafi precedenti, i vari aspetti da considerare per l'implementazione del sistema Thereminux, descriviamo in questa sezione alcune tecnologie esistenti che sono state sfruttate all'interno del progetto.

3.7.1 La scheda *Beagle Board* e l'API USB Gadget del kernel Linux

Esistono in commercio numerose soluzioni hardware per lo sviluppo di applicazioni embedded di più o meno grande complessità. La maggior parte di queste mettono a disposizione un microcontroller, una certa quantità di memoria e alcune periferiche per l'I/O. Per l'applicazione descritta in questo documento l'attenzione a tali soluzioni è principalmente dovuta al fatto che esse offrono un'ottima opportunità per risparmare tempo nella progettazione e sviluppo hardware, per potersi concentrare così sul software. Tra quelle più semplici ed economiche, vale sicuramente la pena menzionare le schede Arduino [36] e Teensy [37]. Entrambe mettono a diposizione un microcontroller Atmel della famiglia AVR, memoria Flash e RAM, un convertitore A/D per la gestione di segnali analogici e, infine, un controller USB per la comunicazione con un pc host. Ad un'analisi più approfondita, però, è emerso che queste soluzioni non erano adeguate all'applicazione qui descritta. Infatti, ad esempio, Arduino non è pensato per permettere la personalizzazione del dispositivo USB fino a ridefinirne le carattestiche strutturali, come il numero e il tipo di endpoint, mentre fornisce un'API già pronta per implementare applicazioni software embedded nascondendo i dettagli dell'hardware a livello, ad esempio, di istruzioni del processore. Questo non vale per Teensy, che invece permette di ridefinire interamente il firmware eseguito sul microcontroller e, quindi, anche le specifiche del dispositivo USB, e fornisce un'applicazione semplicissima lato host per il caricamento di tale firmware sul dispositivo. In questo caso, purtroppo, il problema è stato riscontrato sulle capacità di

calcolo della scheda: infatti il microcontroller AT90USB1286 montato sul Teensy mette a disposizione un convertitore A/D ad approssimazioni successive con 10 bit di risoluzione e con un massimo di performance garantita di 200 KHz, cosa che (cfr. 3.1), fa capire come tale controller è troppo lento per la nostra applicazione, in cui si richiede un campionamento del segnale sonoro in tempo reale a 44.1 KHz.
Dunque, ci si è spostati su un prodotto computazionalmente più potente. Dopo un'attenta ricerca, la Beagle Board [38] è sembrata fare al caso nostro (Figura 3.17).

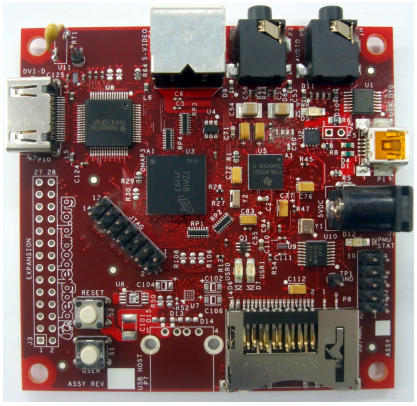

Figura 3.17: La Beagle Board. Fonte [16]

Questa scheda, infatti, oltre a mettere a disposizione risorse hardware più potenti (come il processore OMAP3530 [14] basato su un microcontroller ARM a 720 Mhz) è specifica per lo sviluppo di applicazioni multimediali: ad esempio fornisce un convertitore A/D ad approssimazioni successive a 16 bit in grado di effettuare campionamenti fino a 48 KHz, mette a disposizione connettori mini-jack standard per le applicazioni audio e anche controller specifici per la gestione e l'elaborazione video. La presenza di tali connettori audio, tra l'altro, è stata fondamentale per permettere il collegamento allo strumento anche di un comune microfono audio, dunque permettendo di posticipare l'implementazione hardware del Theremin analogico.
Naturalmente, però, per programmare e gestire una scheda di questa portata non è più sufficiente soltanto l'utilizzo di un firmware che possa accedere ai registri e ai pin del processore: infatti, la Beagle Board viene solitamente utilizzata con un sistema operativo vero e proprio e, in particolare, è compatibile con Linux Embedded [39]. In particolare, la sequenza di avvio della Beagle dalla sua memoria flash si articola in tre fasi: si avvia prima il tool *X-Loader* [40], che si occupa di avviare il firmware *U-Boot* [41]. Questo fornisce un'interfaccia sul dispositivo dall'host tramite il bus seriale, permettendo di

eseguire diversi comandi per il debug hardware e, inoltre, può avviare un'immagine Linux passandogli, volendo, anche opportuni parametri.
A questo punto, avendo a disposizione un kernel Linux, l'utilizzo della scheda Beagle Board nella nostra applicazione si traduce sostanzialmente nella necessità di scrivere un device driver USB lato periferica, che sia in grado di inizializzare e gestire correttamente l'hardware e la comunicazione con l'host. Per questo, il kernel Linux mette a disposizione l'interfaccia *USB Gadget*.

L'API *USB Gadget*

Questa interfaccia, definita all'interno del sorgente del kernel in *include/linux/usb/gadget.h*, non viene solitamente trattata nel normale sviluppo di device driver USB, in quanto, nella maggior parte dei casi, questi vengono eseguiti sul sistema host, quindi viene delegata la gestione del protocollo vero e proprio al *core* USB del kernel. Nel caso di sistemi che vengono eseguiti sulla periferica USB, invece, è necessario poter dare una definizione di tutti i parametri necessari alla definizione del device USB (cfr. 3.2.3), come gli endpoint, le interfacce, i descrittori, ecc.
Oltre al sorgente dell'interfaccia, all'interno del kernel (cfr. *drivers/usb/gadget/*) sono presenti numerosi esempi di driver gadget che implementano svariate funzionalità. Molti di questi (come il semplice *gadget_zero*) utilizzano il framework *composite* (cfr. *include/linux/usb/composite.h*) che è pensato in particolare per i dispositivi USB *composite* (cfr. 3.2.2). Per la nostra applicazione, comunque, basterà una sola configurazione e una sola interfaccia che utilizza diversi endpoint, per cui ci si è basati su driver che non utilizzato tale framework, come *file_storage* (cfr. *drivers/usb/gadget/file_storage.c*) o *gmidi* (cfr. *drivers/usb/gadget/gmidi.c*).
Sostanzialmente, per la scrittura di tali driver si procede come segue: come molto spesso avviene nei device driver, viene mantenuta una struttura generale di descrizione di tutte le caratteristiche del dispositivo:

```
struct my_device {
        spinlock_t              lock;
        struct usb_gadget       *gadget;
        struct usb_request      *req;
        u8                      config;
        struct usb_ep           *in_ep, *out_ep;
        /* ... */
}
```

Qui si nota subito la presenza delle strutture `usb_gadget` (che contiene la rappresentazione dei dispositivi gadget), `usb_request` (con le informazioni riguardo le richieste USB che arrivano al dispositivo dall'host) e `usb_ep` (che rappresenta gli endpoint). Come è chiaro, dunque, questo dispositivo definisce solo un endpoint di input e uno di output. Oltre a questa struttura, il driver deve definire le stringhe statiche relative alle informazioni di base sul dispositivo (vendorId, productId, ecc) e tutti i vari descrittori USB specifici per esso. Questi, di solito definiti all'interno del file *include/linux/usb/ch9.h*, rappresentano le informazioni standard descritte nel capitolo 9 del documento di specifiche dell'USB [3], affinché ogni dispositivo USB-compliant le adoperi (cfr. 3.2.3). I principali descrittori sono:

- `device_descriptor`: contiene informazioni generali sul dispositivo, tra cui il numero di configurazioni utilizzate;
- `config_descriptor`: descrive una configurazione specifica (ad esempio specificando il numero di interfacce che contiene);
- `interface_descriptor`: descrive una interfaccia (indice, numero di endpoint contenuti, ecc);
- `endpoint_descriptor`: rappresenta un endpoint, conservandone tutte le informazioni peculiari, come la direzione di trasferimento dei dati (`USB_DIR_IN` o `USB_DIR_OUT`) e gli attributi dell'endpoint (come il tipo);
- `usb_descriptor_header`: un contenitore per i vari descrittori sopra dichiarati, in modo da descrivere facilmente la *funzione* implementata dal dispositivo (da ricordare che, nella specifica USB, i dispositivi periferici vengono definiti anche, appunto, "funzioni", cfr. 3.2);
- `usb_string`: struttura di comodità (definita nell'interfaccia *include/linux/usb/gadget.h*) per definire un descrittore stringa che comprende un id e la stringa di caratteri vera e propria;
- `usb_gadget_strings`: contenitore per il raggruppamento di un array di stringhe espresse nella stessa lingua (ad esempio, come si legge nel sorgente dell'API, si può specificare `0x0409 for en-us`).

In seguito, dopo aver dichiarato tali descrittori, viene definita la `struct usb_gadget_driver`, che contiene i puntatori alle principali funzioni che verranno richiamate dal kernel nel momento opportuno. In particolare, alcune di queste rivestono particolare interesse e verranno descritte tra poco. Come

tutti i moduli del kernel, anche i driver gadget hanno una funzione di `__init` e una di `__exit`. All'interno della prima, solitamente, viene registrata la struttura `usb_gadget_driver` tramite la funzione `usb_gadget_register_driver()`; prima che questa ritorni, il kernel richiama la funzione `bind()` definita dal driver. È a questa che spetta l'inizializzazione delle risorse del driver, come gli endpoint. A questo proposito, il file *drivers/usb/gadget/epautoconf.c* fornisce alcune funzioni di utilità per la configurazione iniziale di tali endpoint, come la `usb_ep_autoconfig()`, che verifica la disponibilità degli endpoint, definiti tramite i vari descrittori, sull'hardware sul quale il driver è eseguito. Una volta terminata la procedura di bind, il gadget risulta attivo e può ricevere richieste sulla sua control pipe (`ep0`). Quando tali richieste (cfr. 3.2.3 e Figura 3.6 per l'elenco completo) arrivano al dispositivo, viene richiamata la funzione `setup()`, che solitamente viene implementata tramite un blocco `switch / case`: a questa funzione viene passata una struttura `usb_ctrlrequest` con le specifiche della richiesta corrente, e una struttura `usb_request` che contiene un buffer in cui inoltrare le risposte. Ad esempio, a fronte della richiesta `USB_REQ_GET_DESCRIPTOR` verrà copiata (tramite una semplice `memcpy`) la struttura con il descrittore richiesto sul buffer dell'endpoint; oppure, per la richiesta `USB_REQ_SET_CONFIGURATION`, l'host richiede al dispositivo di rendere attiva la configurazione presentatagli, cosa che impone al dispositivo di attivare gli endpoint allocati (chiamando la funzione `usb_ep_enable()`). Infine, al termine delle varie richieste, il dispositivo inoltra i dati di risposta all'host tramite la funzione `usb_ep_queue()`.
Se le funzioni `bind()` e `setup()` permettono di rendere attivo il gadget, naturalmente a queste dovranno essere fatte corrispondere opportune funzioni di `unbind()` e `disconnect()` per liberare le risorse. In particolare, la `disconnect()` ripulirà la configurazione corrente disabilitando i vari endpoint dichiarati (tramite opportune chiamate alla funzione `usb_ep_disable()`). Dopo questa chiamata, può essere avviata una nuova `setup()` per impostare una nuova configurazione, oppure potrà essere richiamata la `unbind()` per effettuare la disconnessione definitiva. Questo, sostanzialmente, farà sì che tutte le risorse del driver vengano liberate.
Infine, la funzione di `__exit` de-registrerà il driver tramite una chiamata alla funzione `usb_gadget_unregister_driver()`.

3.7.2 Il kernel *real time*

Il sistema software del Thereminux, sia lato host che lato dispositivo, deve garantire di mantenere dei valori di latenza più bassi possibile, come specificato anche dai requisiti RSW_NF.2 e RSW_NF.3. Per soddisfare questa necessità,

è stata utilizzata una patch disponibile e aggiornata per tutte le versioni stabili del kernel (fino alla 2.6.33 al momento della scrittura di questo documento [56]), chiamata patch `CONFIG_PREEMPT_RT`, che definisce quello che viene chiamato *Kernel Real-Time* [42] [43]. Per applicazioni specificatamente audio, esistono dimostrazioni [45] che mostrano come questo kernel possa raggiungere un tempo di latenza inferiore ai 3ms.
La patch, scritta e mantenuta da Ingo Molnar, è basata sull'assunzione che, per ottenere i migliori risultati possibili in termini di latenza del kernel, bisogna massimizzare la quantità di codice *preemtible* interno al kernel, cercando di cambiare il meno possibile in termini di codice. Per questo, vengono introdotte diverse caratteristiche non native:

- tutte le primitive di locking interne al kernel (semafori, *spinlock*, *seqlock*, ecc) vengono rese *preemptible*. Questo crea alcune sostanziali differenze rispetto al comportamento "normale": ad esempio, uno *spinlock* viene di solito acquisito in contesti atomici, ovvero in cui chi possiede il lock esegue le sue istruzioni rapidamente, senza andare in *sleep* e senza essere interrotto (a parte in caso di interrupt se non è stata usata la funzione `spin_lock_irqsave()`), anche perché i processi che rimangono bloccati su quel lock implementano un'attesa attiva su di esso. Permettere che il processo che detiene lo spinlock possa perdere il controllo del processore anche all'interno della sezione critica fa sì che alcune modifiche siano imposte: ad esempio, i processi in attesa su quel lock possono andare in *sleep*. Naturalmente non viene intaccata la sicurezza della sincronizzazione, visto che comunque tutti i processi sono sincronizzati sul lock, però diventa importante che il lock non sia preso quando sono stati disabilitati gli interrupt o la funzione di *preemption*, in quanto porre un task in *sleep* e ripristinarlo crea un overhead aggiuntivo che non è giustificabile se il tempo di mantenimento del lock è troppo breve [46];

- visto che tali sezioni critiche diventano *preemptible*, anche l'uso delle cosiddette variabili *per-CPU* necessita una maggiore cura: infatti è possibile che un processo, che si trova all'interno di una sezione critica in cui sta per aggiornare la variabile per-CPU, perda il controllo della CPU e venga messo in scheduling su un'altra CPU, il che farà sì che aggiornerà la variabile sbagliata, visto che, come noto, una variabile per-CPU viene mantenuta in diverse copie per i diversi processori. Per evitare ciò, in questo caso, è necessario disabilitare esplicitamente la *preemption* per il tempo necessario all'aggiornamento della variabile. Questo può essere fatto utilizzando la funzione `get_cpu_var()` o la

`preempt_disable()`, oppure usando un lock specifico per le variabili per-CPU;

- nel caso in cui si rende necessario utilizzare gli spinlock nel modo tradizionale è sempre possibile dichiararli come `raw_spinlock_t` e utilizzando la stessa API dei classici `spinlock_t`. La patch è stata progettata per implementare una sorta di *method overridding*, in cui le medesime funzioni si comportano nella maniera "classica" se gli viene passato un `raw_spinlock_t`, mentre eseguono il locking *preemptive* in caso di `spinlock_t`. Come si legge sulla documentazione, però, l'uso dei `raw_spinlock_t` deve rappresentare un'eccezione valida solo in determinati contesti particolarmente critici, come lo scheduler o alcune porzioni di codice che hanno a che fare direttamente con l'hardware;

- viene implementata *priority inheritance* per gli spinlock e i semafori interni al kernel: per comprendere questo, occorre prima spiegare il fenomeno della *priority inversion*. Supponiamo che un processo A con bassa priorità detenga un lock L1, e che arrivi un processo B con media priorità a chiedere il controllo del processore. Poiché il lock è *preemptible*, il processo A cederà il processore a B. Supponiamo poi che un processo C ad alta priorità cerchi di acquisire il lock su L1: C si bloccherà su questa richiesta, ma il processo A non potrà terminare la sua sezione critica per diverso tempo in quanto bloccato dal processo B. Per questo si parla di *priority inversion*, perché il processo con maggiore priorità è costretto ad aspettare quello con priorità minore. Per evitare questo fenomeno, la *priority inheritance* fa sì che, in uno scenario come quello descritto, il processo A prenda, soltanto per il tempo necessario a concludere la sua sezione critica e quindi a rilasciare il lock, i privilegi del processo C. In questo modo, sempre nell'esempio esposto, A potrebbe riconquistare il processore (in quanto appare come un processo di priorità maggiore a B) e rilasciare il lock il prima possibile. Questo, comunque, crea dei problemi nel caso si usino lock read/write in cui ci sono molti lettori che detengono un lock e un solo scrittore. In questo caso bisognerebbe aumentare i privilegi e poi ripristinarli per numerosi processi, cosa piuttosto inefficente. Per questo, la politica applicata con la patch è che un solo lettore alla volta possa prendere un certo lock, cosa che riduce la scalabilità ma mantiene un certo grado di efficienza;

- gli interrupt vengono convertiti in normali thread del kernel, dunque diventano anch'essi *preemptible*. L'utilizzo delle classiche funzioni di disabilitazione degli interrupt, come la `spin_lock_irqsave()`, non disabilitano più gli interrupt hardware. L'unico caso in cui viene preserva-

to il contesto di esecuzione *interrupt* è quando viene specificato il flag `SA_NODELAY` alla creazione dell'interrupt: in questo caso è necessario utilizzare il comportamento di locking tradizionale, ad esempio tramite la `raw_spinlock_t`. Comunque, anche l'uso del flag `SA_NODELAY` viene utilizzato soltanto da parti del kernel molto specifiche e critiche, dunque non deve essere considerato una cosa comune;

- è stata utilizzata una nuova infrastruttura per la gestione dei timer, in modo da garantire una maggiore risoluzione in caso di timeout o altri eventi con richieste stringenti di tempo.

Installazione e utilizzo della patch

L'utilizzo della patch è abbastanza semplice. In pratica si richiede, dopo averla applicata (tramite il comando `patch -p1`) di effettuare le seguenti configurazioni:

- `CONFIG_PREEMPT_RT` abilitato;
- High-Resolution Timer (HR Timer) abilitato;
- disabilitare opzioni di power management come ACPI o APM.

Successivamente è necessario ricompilare il kernel. Fatto questo, bisogna definire alcuni parametri affinché applicazioni specifiche nello user-space possano usufruire delle priorità di scheduling real time nel kernel (che altrimenti sono richiamabili esplicitamente soltanto dal kernel stesso). Infatti il kernel assegna ai processi due valori di priorità: quello "regolare" (impostabile con la system call `nice`) che va da -20 (priorità massima) a +19 (priorità minima), e quello *real-time*, che va da 1 (priorità più alta) a 99 (priorità più bassa) [5]. Usando il kernel real-time può essere importante far sì che alcune applicazioni nello user space possano essere eseguite con determinati valori di priorità real-time, oltre che al valore di priorità regolare.
Per fare questo, è possibile modificare il file */etc/security/limits.conf*, ma esiste il tool open-source `set_rlimits` per eseguire questa operazione in maniera più "user friendly" [47]. Attraverso questa utility, è possibile impostare manualmente alcuni importanti parametri relativi all'esecuzione di specifiche applicazioni da parte di specifici utenti. Ad esempio, il file di configurazione */etc/set_ rlimits.conf* potrebbe contenere una riga come la seguente:

```
@daniele /usr/bin/jackd nice=-10 rtprio=95 memlock=250000
```

In questo modo, si specifica che un utente (`@daniele`) può eseguire un programma (`/usr/bin/jackd`) con opportuni valori di priorità *nice* (`nice=-10`) e *real-time* (`rtprio=95`). Infine, si specifica che questo programma potrà ottenere dei lock su fino a 250 MB di memoria (`memlock=250000`). Dopo aver eseguito questa configurazione su tutti i programmi desiderati, sarà possibile avviarli con le priorità real-time, da utente senza privilegi, tramite il comando `set_rlimits <nome-programma>`.

3.7.3 Implementazioni per la *pitch detection*

Come trattato precedentemente (cfr. 3.6), al fine di implementare la conversione Voltage-to-MIDI, è stato necessario utilizzare un algoritmo di pitch detection. Tra i progetti open source che forniscono implementazioni già fatte di questa funzionalità, salta all'occhio *aubio* [48]. Questo implementa una libreria nello user space per l'annotazione di informazioni relative ai segnali audio. Tra le funzionalità che implementa ci sono filtri digitali, riconoscimento di tempo e, ovviamente, la pitch detection. Ci sono diversi metodi implementati per quest'ultima ma, anche in base alle valutazioni sull'efficienza di tali metodi, quello più interessante da tenere in considerazione è lo *YIN*. Il codice del metodo è racchiuso in un'unica funzione (cfr. file *aubio_src/pitchyin.c*):

```
aubio_pitchyin_do (aubio_pitchyin_t * o, fvec_t * input,
                   fvec_t * out)
{
   smpl_t tol = o->tol;
   fvec_t *yin = o->yin;
   uint_t j, tau = 0;
   sint_t period;
   smpl_t tmp = 0., tmp2 = 0.;
   yin->data[0] = 1.;
   for (tau = 1; tau < yin->length; tau++) {
     yin->data[tau] = 0.;
     for (j = 0; j < yin->length; j++) {
       tmp = input->data[j] - input->data[j + tau];
       yin->data[tau] += SQR (tmp);
     }
     tmp2 += yin->data[tau];
     yin->data[tau] *= tau / tmp2;
     period = tau - 3;
     if (tau > 4 && (yin->data[period] < tol) &&
```

```
        (yin->data[period] < yin->data[period + 1])) {
      out->data[0] = fvec_quadint (yin, period);
      goto beach;
    }
  }
  out->data[0] = fvec_quadint (yin, fvec_min_elem (yin));
beach:
  return;
}
```

Sono state definite diverse strutture e costanti all'interno del codice di aubio. Ad esempio, `smpl_t` è definito come `float`, mentre `uint_t` e `sint_t` sono definiti, come intuibile, rispettivamente come `unsigned int` e (signed) `int`. Oltre a questo, la struttura `fvec_t` è definita come segue:

```
typedef struct {
    uint_t length;
    smpl_t *data;
} fvec_t;
```

Con `length` la lunghezza del buffer e `data` l'array di campioni del segnale. Inoltre, vediamo come argomento della funzione una struttura di tipo `aubio_pitchyin_t`, definita tramite `typedef` dall'analoga `_aubio_pitchyin_t`:

```
struct _aubio_pitchyin_t
{
  fvec_t *yin;
  smpl_t tol;
};
```

Oltre all'array di campioni `yin`, la struttura contiene un valore di *tolerance*, una soglia passata la quale il risultato si può considerare trovato (di solito, come si legge nel sorgente di aubio, questo valore viene impostato su .85).
La funzione esegue il calcolo della formula mostrata nel paragrafo 3.6.1, calcolando incrementalmente il valore di $d'_t(\tau)$ da quello di $d_t(\tau)$. Infine, tramite la funzione `fvec_quadint()` ritorna l'indice del picco del segnale (che rappresenta il pitch del suono) calcolato tramite interpolazione quadratica.
Come vedremo più avanti (cfr. 6), questo codice è stato utilizzato per sviluppare il convertitore pitch-to-midi importandolo a livello del kernel. Infatti, anche se l'algoritmo richiede una certa quantità di calcoli floating point (e questo sembrerebbe non rientrare nelle buone pratiche di programmazione del kernel) è chiaro che un'applicazione come questa, che richiede calcoli di

elaborazione del segnale, necessiti dell'uso di tale tecnica. Naturalmente, però, sono state previste delle precauzioni, seguendo quanto descritto in [5] [57]: tutte le operazioni critiche sono state implementate direttamente per essere eseguite nella Floating Point Unit (FPU) hardware presente nella cpu: per fare questo, il modulo del kernel sviluppato è stato compilato con le opzioni `-ffast_math` e `-mhard_float` e, in fase di linking, il driver controlla esplicitamente se la cpu su cui è in esecuzione è in possesso di tale unità hardware (usando la macro `cpu_has_fpu` definita nel file */include/asm/cpufeature.h*), pena l'impossibilità per il modulo di essere caricato. In ogni caso, l'inefficienza dell'esecuzione di istruzioni floating point nel kernel sta nel fatto che la FPU è un modulo hardware separato dalla ALU della cpu, per cui è necessario salvare e ripristinare il contesto dei registri della FPU ad ogni context switch. Questa grave inefficienza può essere risparmiata preservando l'esecuzione tramite le istruzioni `kernel_fpu_begin()` e `kernel_fpu_end()`: esse, infatti, permettono al thread corrente del kernel di dichiarare esplicitamente che ha la necessità di utilizzare la FPU, per cui c'è bisogno di effettuare il salvataggio e ripristino del contesto. Quando tali funzioni non sono inserite, questa dispendiosa operazione non viene fatta.

Capitolo 4

Cenni sulla progettazione del sistema hardware

Dopo aver trattato nel Capitolo 3 lo studio delle questioni maggiormente critiche del sistema, elaborandone le varie possibilità e definendo soluzioni ritenute appropriate, in questo capitolo e nel seguente (cfr. Capitolo 5) verrà descritto il sistema vero e proprio. Per prima cosa, in questo capitolo verrà data la descrizione del sistema hardware, ovvero la composizione degli strumenti hardware messi a disposizione dalla Beagle Board, tramite uno schema a blocchi (cfr. 4.1).

4.1 Diagramma a blocchi

In Figura 4.1 viene mostrato il diagramma a blocchi che descrive il sistema hardware usato.

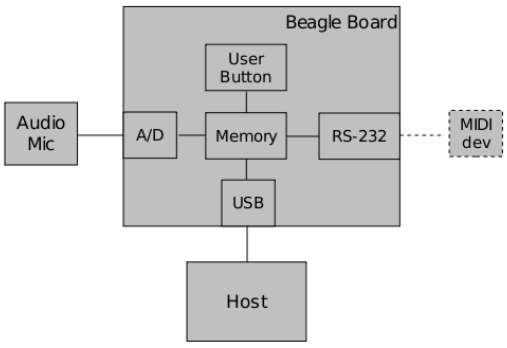

Figura 4.1: Diagramma a blocchi del sistema hardware del Thereminux

Un microfono audio è stato collegato alla Beagle Board tramite cavo audio standard con connettore di tipo *jack*. Da qui, il segnale viene convertito tramite il modulo A/D presente sulla scheda (16 bit, 44.1 KHz) e i campioni bufferizzati all'interno della memoria, dalla quale poi vengono inviati al sistema host tramite USB. Oltre a questo, come specificato nel requisito RHW_F.4, il sistema dispone di un'uscita MIDI-out tramite la quale sarebbe possibile controllare altri strumenti MIDI. Questa caratteristica è stata predisposta utilizzando l'interfaccia RS-232 presente sulla scheda: infatti, il MIDI è essenzialmente un protocollo seriale, anche se occorre prestare qualche attenzione riguardo il collegamento di un circuito MIDI ad un cotroller seriale RS-232. I dispositivi RS-232 [49], infatti, lavorano su due livelli di voltaggio (da -3 a -25 volt per il livello negativo e da +3 a +25 volt per quello positivo). Effettuare una conversione tra segnale RS-232 e MIDI è quindi non banale: per prima cosa, occorre programmare opportunamente il baud-rate di trasmissione seriale secondo lo standard MIDI, ovvero a 31.250 bit/secondo. Questo, comunque, non è un grosso problema, in quanto, come riferito in [16], la Beagle Board permette di programmare il baud-rate delle sue trasmissioni seriali.
Inoltre, il segnale fornito dal protocollo MIDI (0 e 5 volt) rispetta lo standard TTL di comunicazione digitale, mentre la RS-232 considera come non definito un segnale che si trova tra i -3 e i +3 volt. La conversione tra i due dovrà dunque essere assegnata ad una apposito modulo hardware (non implementato). In questa fase, il sistema di invio di messaggi MIDI-out tramite il Thereminux è stato implementato in software e testato utilizzando delle utility per la connessione seriale con il sistema host (come *minicom*).
Infine, come richiesto dal requisito RHW_F.5, il sistema deve fornire elementi hardware programmabili tramite opportuni messaggi dal sistema host. Per implementare questa funzionalità è stata sfruttata una caratteristica già integrata all'interno della Beagle Board. Infatti, la scheda viene fornita con un pulsante hardware programmabile dall'utente. Per gli scopi di test del prototipo del Thereminux, questo è sembrato essere già sufficiente.

Capitolo 5

Progettazione del sistema software

In questo capitolo verranno trattate tutte le questioni relative alla progettazione effettuata per l'implementazione del sistema software. In questo senso, si è cercato di documentare al meglio varie prospettive (quella dell'utente che interagisce, quella del dispositivo, quella della struttura software, ecc) al fine di analizzare ogni aspetto e cercare di ottenere un buon design. L'approccio che verrà utilizzato è quello *bottom-up* dal punto di vista dell'utente: si partirà con lo studiare le possibili interazioni dell'utente con il sistema (cfr. 5.1), dando una prima bozza dei flussi di base che avvengono; partendo da queste, si analizzerà il comportamento del sistema a fronte delle varie interazioni, dando un modello generale degli stati possibili (cfr. 5.2) dalle diverse prospettive del sistema host e del dispositivo. In seguito, si darà la descrizione strutturale del sistema, specificandone i vari "oggetti" e le interazioni che intercorrono tra di essi (cfr. 5.3). Infine, si darà una descrizione dei principali flussi che avvengono all'interno del sistema, mappandoli sulla struttura fornita (cfr. 5.4).
Per la descrizione di seguito, si farà un ampio uso di diagrammi UML, in modo da fornire rappresentazioni grafiche e maggiormente intuitive del design.

5.1 L'interazione con l'utente: *use cases*

Una prima prospettiva utile alla comprensione delle funzionalità del sistema è l'interazione che l'utente può avere con esso. I requisiti precedentemente descritti (cfr. Capitolo 2) hanno già dato una lista di quali sono queste interazioni, ma può essere utile usufruire di un diagramma dei *casi d'uso* per

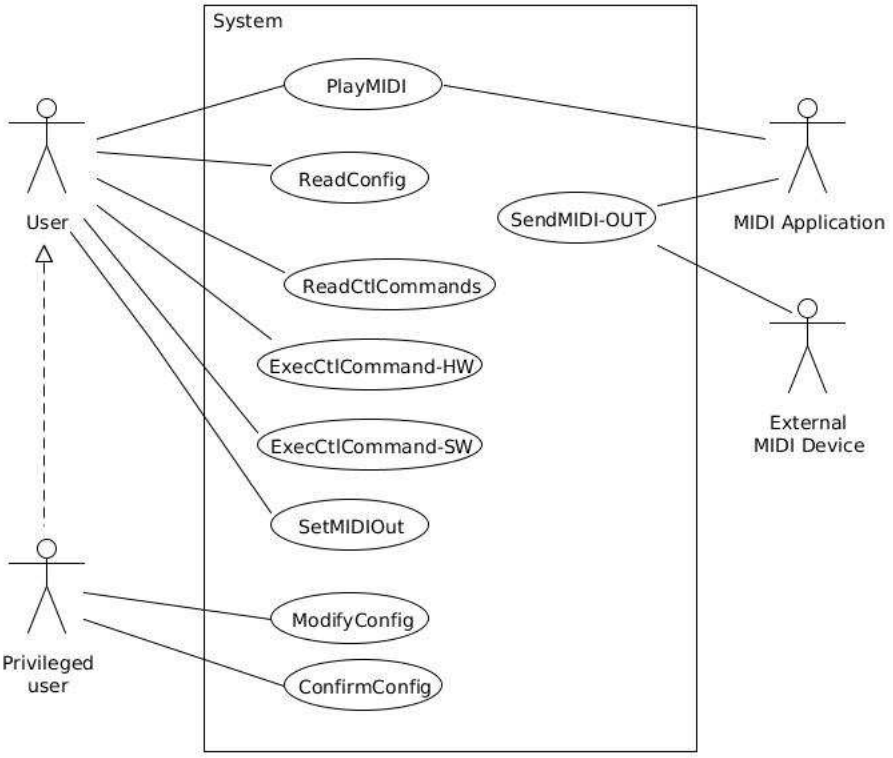

Figura 5.1: Diagramma dei casi d'uso per il sistema

ottenerne una visione migliore. La Figura 5.1 mostra quali sono i casi d'uso con cui l'interazione avviene.
Ogni caso d'uso rappresenta un flusso di azioni che parte da un input dell'utente. Come è possibile vedere, vengono mostrati utenti "umani" e non. Ad esempio, le applicazioni MIDI che girano nello user space del sistema host vengono considerate utenti, in quanto interagiscono con il sistema come tali. Questo vale anche per i dispositivi MIDI esterni che possono essere collegati al sistema tramite la porta hardware di MIDI-out (al momento non implementata). Gli utenti umani sono di due tipi: utente non privilegiato o privilegiato. Quest'ultimo può effettuare sul sistema tutte le azioni concesse al primo, da cui è mostrato come "estendere" l'utente non privilegiato, ereditandone così le possibilità di interazione. Il *sistema* a cui si fa riferimento all'interno del diagramma contiene sia gli aspetti del sistema software che di quello hardware: infatti, come si nota, sono presenti elementi di interazione sia del primo che del secondo tipo. Per chiarezza, comunque, nel seguito si farà riferimento al *sistema embedded* (o semplicemente *gadget*) quando si parla

del software che gira sul dispositivo hardware, al *sistema hardware* quando si parla di elementi hardware e, infine, di *sistema host* per riferirsi al sistema software in esecuzione sull'host. In ultimo, forniamo di seguito (cfr. 5.1.1) una lista di tabelle che danno i dettagli, sempre a livello qualitativo, dei flussi eseguiti per ogni caso d'uso.

5.1.1 Specifiche dei casi d'uso

Nome Caso d'uso	**PlayMIDI**
ID	CU.1
Attori	`User`, `MIDI Application`
Descrizione	Permette all'utente `user` di suonare note MIDI tramite lo strumento hardware; queste note MIDI vengono catturate da applicazioni MIDI in ascolto nello user space e suonate.
Precondizioni	Il sistema deve essere attivo e inizializzato (dispositivo hardware acceso e collegato all'host, moduli del kernel caricati e pronti).
Flusso principale	1. L'utente inizia a suonare lo strumento hardware; 2. Il sistema embedded converte il segnale analogico in stringa binaria; 3. Il sistema embedded invia il risultato della conversione sul bus USB; 4. Il sistema host effettua una conversione Voltage-to-MIDI del segnale in ingresso; 5. Il messaggio MIDI viene inviato al livello applicativo.
Flussi alternativi	-
Postcondizioni	Il sistema host ha inviato la nota MIDI, relativa al segnale di input, allo user space.

Nome Caso d'uso	**ReadConfig**
ID	CU.2
Attori	`User`
Descrizione	Permette all'utente `user` di leggere il profilo di configurazione.
Precondizioni	Il sistema deve essere stato inizializzato (profilo di configurazione caricato).
Flusso principale	1. L'utente `user` esegue il comando di lettura del profilo di configurazione. 2. Il sistema host legge il profilo di configurazione e lo stampa in forma tabulare.
Flussi alternativi	-
Postcondizioni	Il sistema host ha stampato correttamente il profilo di configurazione per l'utente.

Nome Caso d'uso	**ReadCtlCommands**
ID	CU.3
Attori	`User`
Descrizione	Permette all'utente di conoscere l'elenco dei comandi di controllo.
Precondizioni	Il sistema host deve essere stato inizializzato (profilo di configurazione caricato).
Flusso principale	1. L'utente esegue il comando di lettura dei comandi di controllo; 2. Il sistema host legge il profilo di configurazione e stampa in forma tabulare i comandi.
Flussi alternativi	-
Postcondizioni	Il sistema host ha stampato correttamente i comandi di controllo per l'utente.

Nome Caso d'uso	**ExecCtlCommand-HW**
ID	CU.4
Attori	`User`
Descrizione	Permette all'utente di eseguire un comando di controllo tramite hardware programmabile.
Precondizioni	Il sistema deve essere inizializzato e attivo (dispositivo hardware acceso e collegato all'host, moduli del kernel caricati e pronti).
Flusso principale	1. L'utente aziona l'hardware programmabile sul dispositivo; 2. Il sistema embedded verifica quale comando di controllo è associato all'hardware azionato; 3. Il sistema embedded esegue il comando di controllo.
Flussi alternativi	-
Postcondizioni	Il sistema embedded ha eseguito il comando di controllo relativo all'hardware azionato.

Nome Caso d'uso	**ExecCtlCommand-SW**
ID	CU.5
Attori	`User`
Descrizione	Permette all'utente di eseguire un comando di controllo tramite opportuni comandi sul sistema host.
Precondizioni	Il sistema deve essere inizializzato e attivo (dispositivo hardware acceso e collegato all'host, moduli del kernel caricati e pronti).
Flusso principale	1. L'utente esegue il comando di attivazione del comando di controllo; 2. Il sistema host verifica la correttezza del comando invocato; 3. Il sistema host invia il comando al sistema embedded; 4. Il sistema embedded esegue il comando di controllo.
Flussi alternativi	Il sistema host verifica che il comando che si vuole eseguire non esiste o non è corretto. Viene mostrato un messaggio di errore all'utente.
Postcondizioni	Il sistema embedded ha eseguito il comando di controllo relativo al comando eseguito sul sistema host.

Nome Caso d'uso	**SetMIDIOut**
ID	CU.6
Attori	`User`
Descrizione	Permette all'utente di attivare/disattivare la possibilità di inviare messaggi MIDI OUT dal sistema host, attraverso lo strumento hardware, verso strumenti MIDI esterni.
Precondizioni	Il sistema deve essere inizializzato e attivo (dispositivo hardware acceso e collegato all'host, moduli del kernel caricati e pronti).
Flusso principale	1. L'utente esegue il comando di attivazione/disattivazione del MIDI OUT; 2. il sistema host modifica la visibilità o meno del dispositivo come device di MIDI-OUT in Alsa; 3. Il sistema host reimposta il profilo di configurazione.
Flussi alternativi	-
Postcondizioni	Il sistema host ha correttamente modificato l'impostazione corrente della funzione di MIDI-OUT.

Nome Caso d'uso	**ModifyConfig**
ID	CU.7
Attori	`Privileged user`
Descrizione	Permette ad un utente privilegiato di modificare liberamente il profilo di configurazione.
Precondizioni	Il sistema host deve essere stato inizializzato (profilo di configurazione caricato).
Flusso principale	1. L'utente `Privileged user` apre il file del profilo di configurazione per la modifica; 2. il sistema host verifica i privilegi dell'utente; 3. Il sistema host permette la modifica del file.
Flussi alternativi	-
Postcondizioni	Il sistema host ha permesso la modifica del file del profilo di configurazione.

Nome Caso d'uso	**ConfirmConfig**
ID	CU.8
Attori	`Privileged user`
Descrizione	Permette ad un utente privilegiato di reimpostare il profilo di configurazione.
Precondizioni	Il sistema deve essere inizializzato e attivo (dispositivo hardware acceso e collegato all'host, moduli del kernel caricati e pronti).
Flusso principale	1. L'utente `Privileged user` esegue il comando per la reimpostazione del profilo di configurazione; 2. il sistema host verifica che il profilo di configurazione sia ben formato e corretto; 3. Il sistema host reimposta il profilo di configurazione.
Flussi alternativi	Il sistema host trova un errore nel profilo di configurazione. Verrà inviata una segnalazione di errore all'utente specificando l'errore riscontrato.
Postcondizioni	Il sistema host ha reimpostato il profilo di configurazione.

Nome Caso d'uso	**SendMIDI-OUT**
ID	CU.9
Attori	`MIDI Application`, `External MIDI Device`
Descrizione	Permette ad un'applicazione nello user space di inviare messaggi MIDI attraverso il dispositivo hardware.
Precondizioni	Il sistema deve essere inizializzato e attivo (dispositivo hardware acceso e collegato all'host, moduli del kernel caricati e pronti).
Flusso principale	1. Un'applicazione invia un messaggio MIDI per la porta MIDI-OUT dello strumento; 2. il sistema host invia il messaggio al dispositivo hardware; 3. il sistema embedded inoltra il messaggio sulla porta MIDI-OUT dello strumento.
Flussi alternativi	Il sistema host trova un errore nel messaggio MIDI da inviare. Verrà inviata una segnalazione di errore all'utente.
Postcondizioni	Il sistema ha inviato correttamente il messaggio MIDI al dispositivo MIDI esterno.

5.2 Analisi tramite macchine a stati

Grazie alla prima analisi effettuata dal punto di vista dell'interazione con gli end-user del sistema sono emerse diverse caratteristiche da approfondire, cercando di soffermare l'attenzione sul sistema vero e proprio. Per fare questo possiamo utilizzare i *diagrammi di stato*. Attraverso tale strumento, è possibile individuare graficamente il ciclo di vita del sistema, dando una prima intuizione dei flussi interni che regolano le diverse attività descritte dai casi d'uso. In questa sezione, quindi, daremo una descrizione generale del ciclo di vita del sistema host e di quello embedded, per poi fornire ulteriori dettagli sui flussi relativi alle principali operazioni da implementare (cfr. 5.2.1).
La Figura 5.2 mostra il ciclo di vita del sistema *host*.

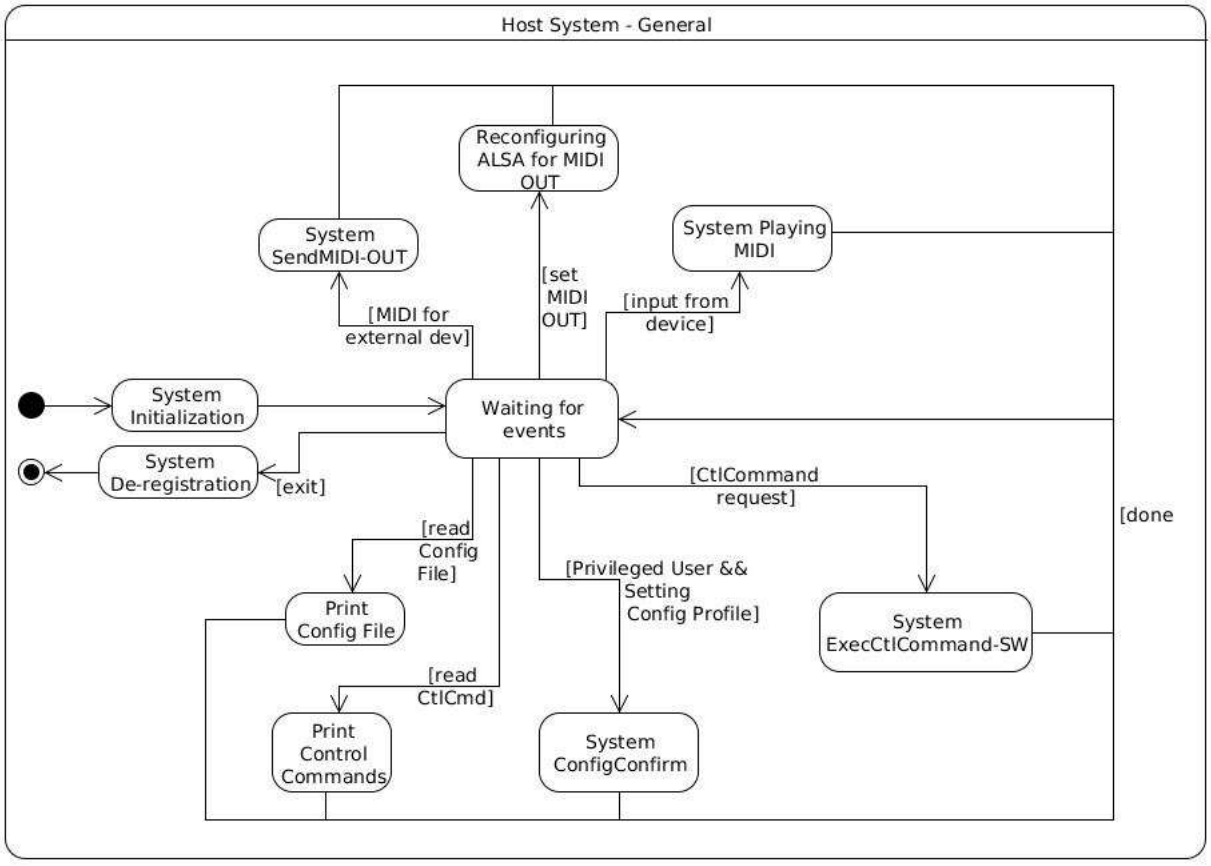

Figura 5.2: Diagramma degli stati che descrive un tipico ciclo di vita del sistema host

Come è facile vedere, il sistema host si basa interamente sul paradigma *event*

driven in quanto, come noto, questa è la normale modalità di esecuzione dei device driver. Gli stati principali descritti in questo diagramma verranno ulteriormente approfonditi più avanti. Tra quelli per i quali manca un diagramma di approfondimento troviamo *Reconfiguring ALSA for MIDI OUT*, che semplicemente inizializza o elimina la funzionalità di MIDI-OUT registrandola o de-registrandola presso ALSA; allo stesso modo, non è stato fornito un approfondimento dei semplici stati *Print Config File* e *Print Control Commands*, in quanto il loro compito è essenzialmente quello di stampare una versione formattata e *user friendly* del contenuto del profilo di configurazione. Infine, anche lo stato *System De-registration* effettua soltanto la liberazione delle risorse, per cui è stato lasciato non ulteriormente specificato a questo livello di descrizione.
Il ciclo di vita del sistema *embedded* è assai simile al precedente (Figura 5.3).

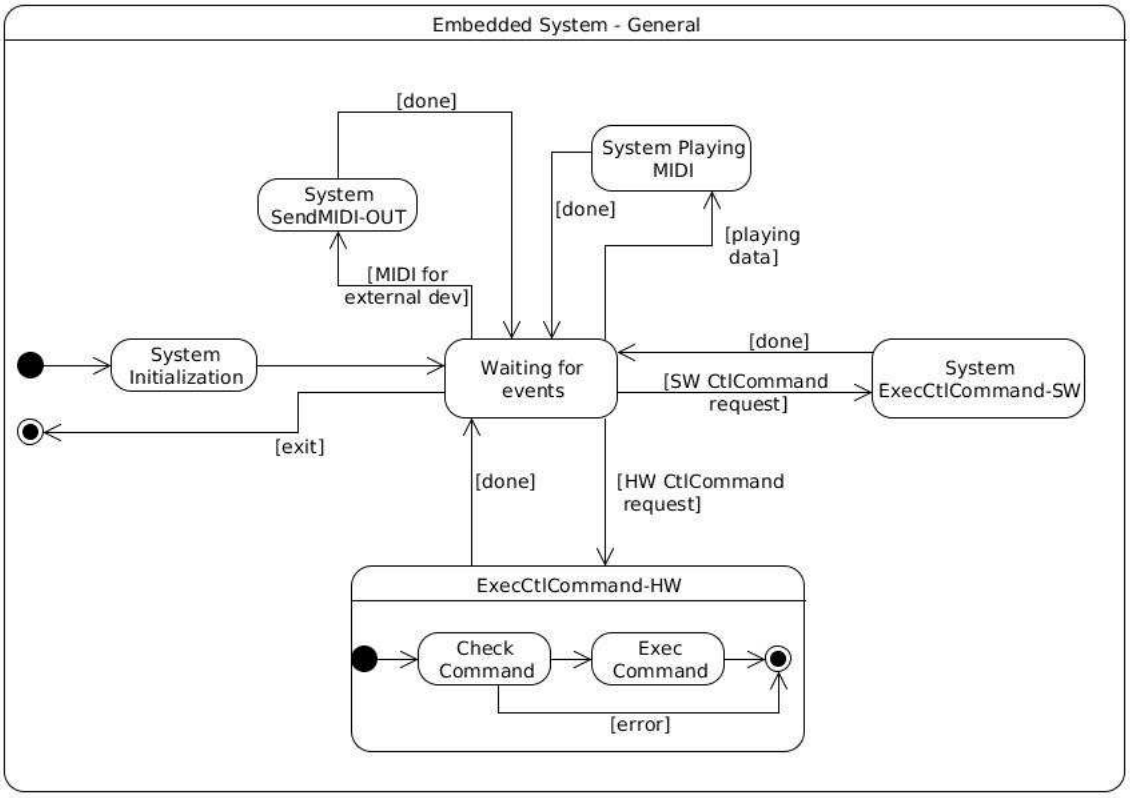

Figura 5.3: Diagramma degli stati che descrive un tipico ciclo di vita del sistema embedded

Qui notiamo molte delle caratteristiche già descritte dal sistema host: nei diagrammi di approfondimento che verranno forniti tra poco, infatti, vedremo come molti degli stati descritti coinvolgono entrambi i lati del sistema. L'unico stato che si è ritenuto importante espandere già in questo diagramma è *ExecCtlCommand-HW* che, come descritto dal caso d'uso `CU.4`, permette di eseguire un comando di controllo azionando un elemento hardware programmabile. La sequenza descritta da questo macro-stato è, comunque, piuttosto semplice, in quanto viene essenzialmente effettuato un controllo sul comando da eseguire e, se questo non è indefinito, viene avviata l'esecuzione.

5.2.1 Dettagli sui principali scenari

Molti degli stati descritti nel precedente paragrafo sono in realtà dei macro-stati che coinvolgono l'azione coordinata di più end-user descritti con i casi d'uso. Vediamone i dettagli.

Inizializzazione del sistema

Per prima cosa, l'inizializzazione. La Figura 5.4 mostra la sequenza di inizializzazione in cui sia il sistema embedded che quello host sono coinvolti.
La linea tratteggiata al centro del macro-stato principale simboleggia l'esecuzione concorrente dei due stati contenuti. In questo caso, il sistema host e quello embedded sono eseguiti, naturalmente, in parallelo, con degli espliciti punti di sincronizzazione dell'esecuzione, corrispondenti agli istanti in cui una delle due macchine avvia una comunicazione con l'altra. Chiaramente, come si può anche constatare consultando il codice dell'implementazione del sistema, lo stato *Sending Setup Data* dà qualitativamente la descrizione del flusso di setup del sistema che, in realtà, coinvolge più interazioni in cui il sistema host richiede le configurazioni al sistema embedded, ne seleziona una e richiede le stringhe che compongono il profilo di configurazione. Per il resto, la sequenza è abbastanza semplice, ma una cosa importante da notare è la presenza dello stato *Setup Config Profile*, che avviene dopo la ricezione dei dati di setup da parte del dispositivo. Infatti, all'interno del dispositivo verrà salvato, sottoforma di più descrittori stringa, un profilo di configurazione di default che risulta funzionante con il dispositivo stesso. In questo modo, alla fine della procedura di inizializzazione, il profilo di configurazione sarà già caricato sul sistema host (per i dettagli sul profilo di configurazione si veda 5.3.1).

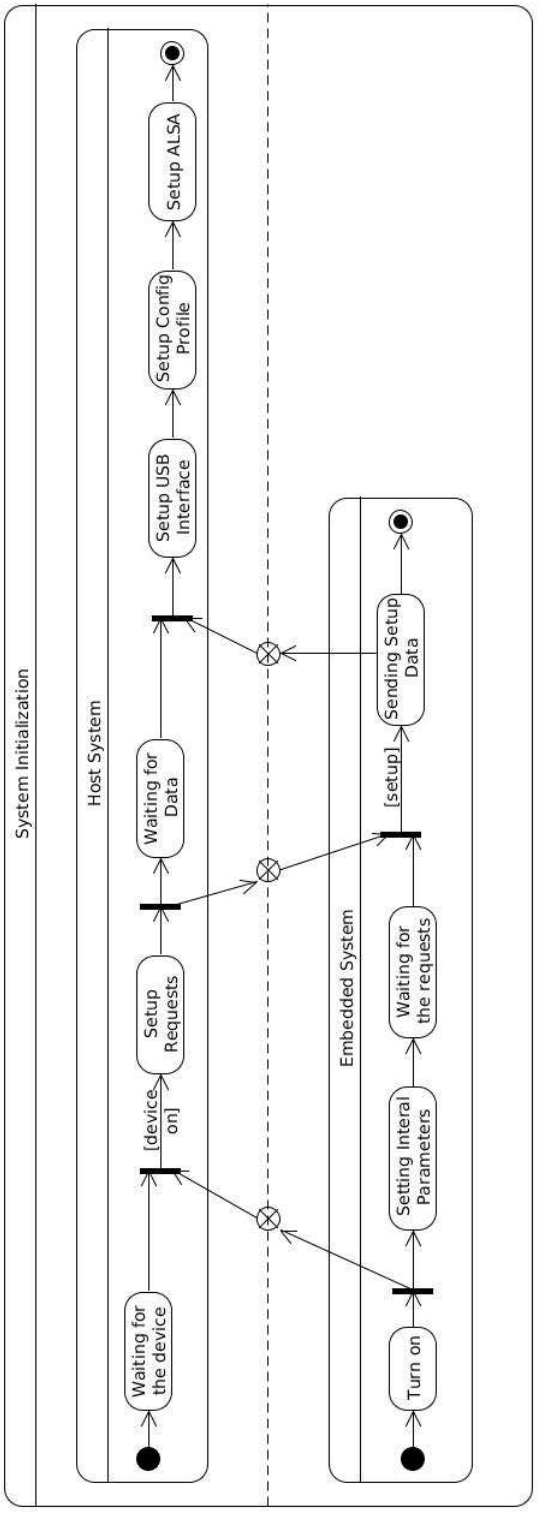

Figura 5.4: Diagramma degli stati che descrive l'inizializzazione del sistema

Suonare note MIDI attraverso lo strumento

La Figura 5.5 mostra lo scenario in cui un utente inizia a suonare con lo strumento hardware.
Come si vede in questo scenario sono presenti tre "attori": il sistema host, il sistema embedded e un'applicazione MIDI che gira nello user space del sistema host. Tutti e tre sono in esecuzione parallela, ma è il dispositivo, ovviamente, che inizia la comunicazione quando il segnale comincia ad essere convertito in digitale. Il sistema host, a questo punto, bufferizza i campioni in ingresso e, dopo la conversione Voltage-to-MIDI, invia questi messaggi all'applicazione in ascolto.

Reimpostare il profilo di configurazione

Altra operazione importante è quella di reimpostare il profilo di configurazione: infatti questa operazione può coinvolgere anche una comunicazione con il sistema embedded, nel caso in cui, nel nuovo profilo di configurazione, sono state effettuate delle modifiche ai dati mantenuti sul dispositivo (ad esempio, le associazioni tra elementi hardware programmabili e comandi di controllo, cfr. 1.1.1 e requisito RSW_F.11). La Figura 5.6 mostra i dettagli di questo scenario.
Da notare il controllo sui privilegi dell'utente chiamante (solo un utente privilegiato, infatti, può effettuare questa operazione) e la correttezza del profilo di configurazione che si vuole impostare, come specificato nei requisiti RSEC_NF.4 e RSEC_NF.9. Inoltre, già lo script di livello applicativo effettua un controllo sul file di configurazione da impostare, per poi delegare il controllo sulla semantica del contenuto al kernel.

Esecuzione di un comando di controllo

Questa operazione, descritta in Figura 5.7, si riferisce allo scenario in cui, dal lato host, viene invocato tramite un opportuno comando dallo user space un comando di controllo dello strumento (cfr. requisito RSW_F.15). Questo scenario, come si vede, è molto simile al precedente, a parte l'assenza del controllo dei privilegi (ogni utente può utilizzare questi comandi) e il fatto che viene sempre stabilita una connessione con il dispositivo hardware per questa operazione, visto che è quest'ultimo che eseguirà fisicamente i comandi di controllo. Anche qui, l'applicazione esegue, prima di passare il controllo al kernel, delle verifiche sui comandi richiesti: se questi sono mal formati o inesistenti si darà un messaggio di errore e non verrà invocato il kernel.

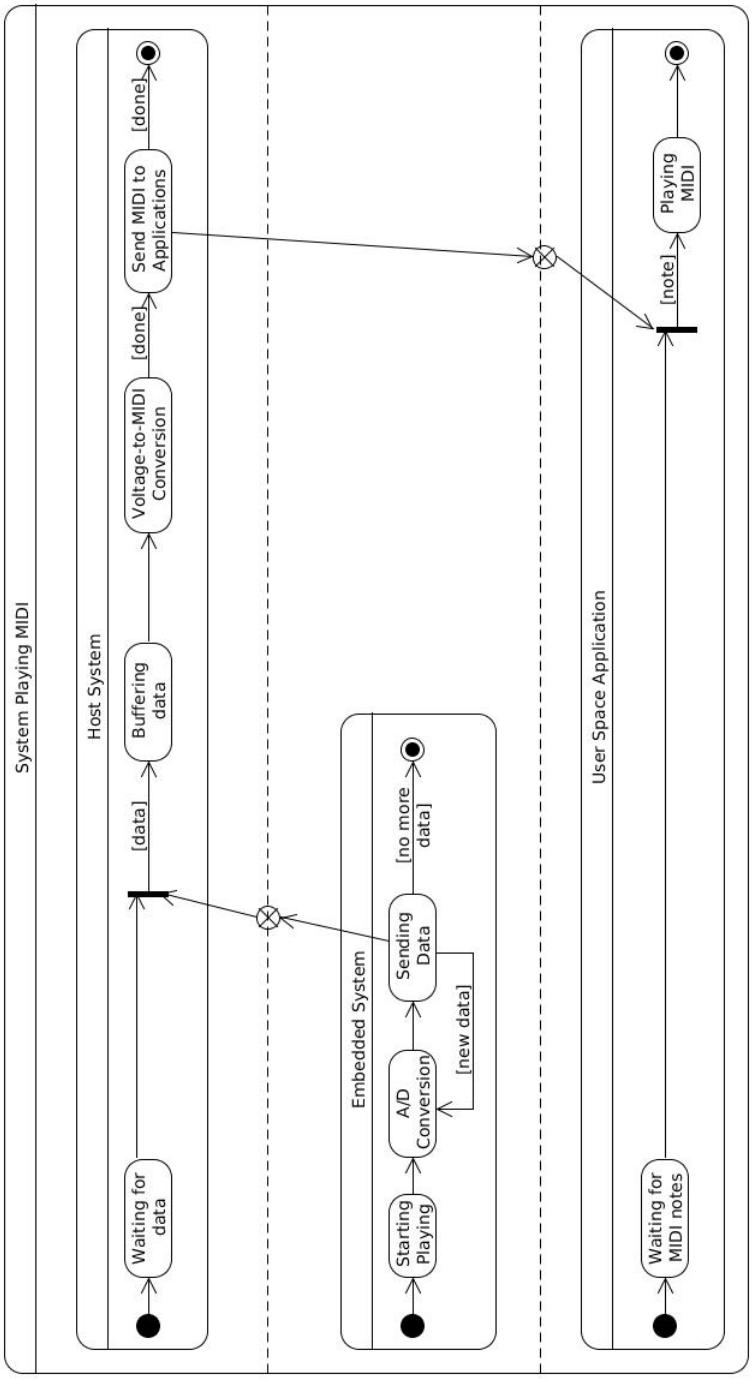

Figura 5.5: Diagramma degli stati che descrive l'esecuzione di note MIDI suonate da parte dello strumento hardware

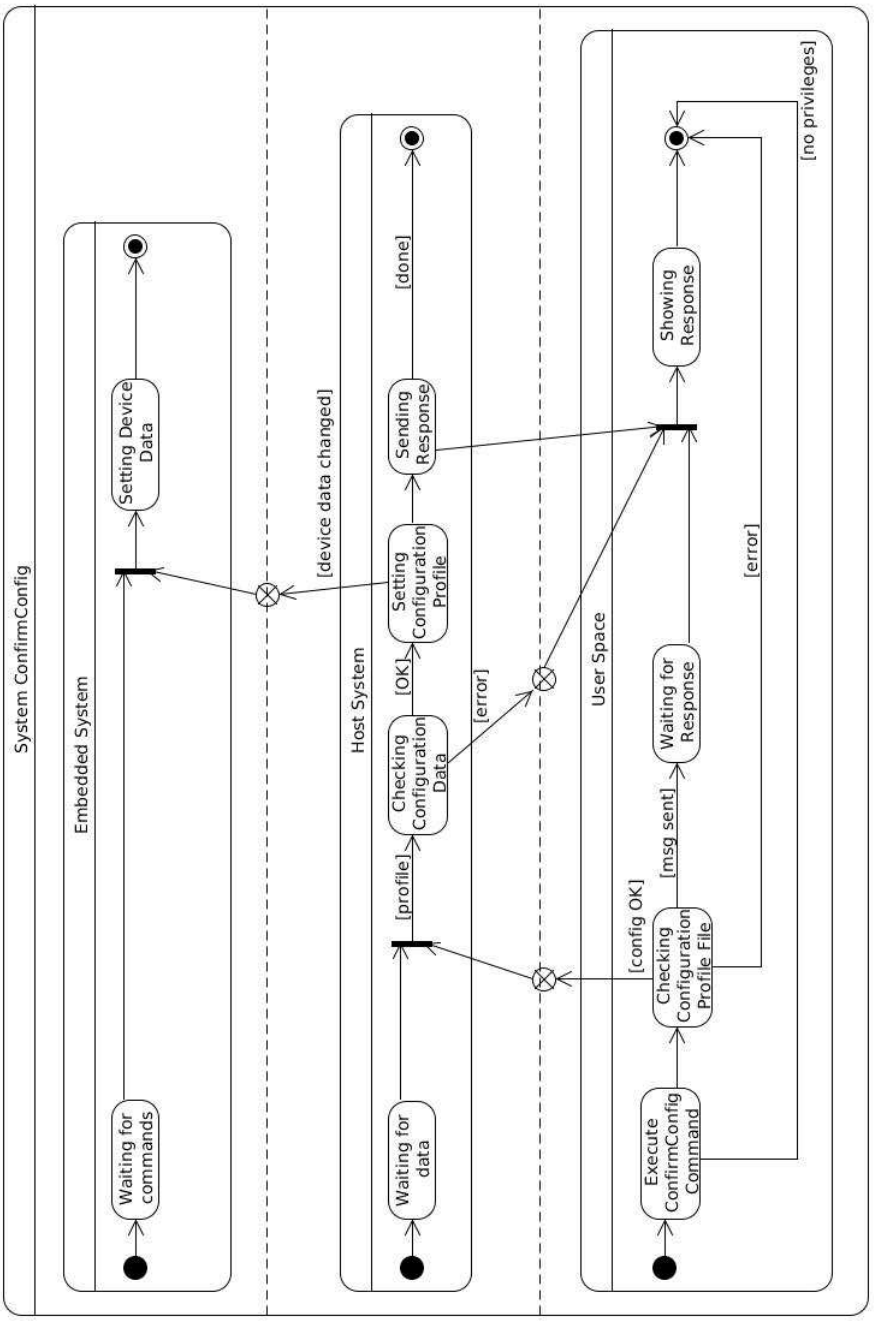

Figura 5.6: Diagramma degli stati che descrive l'operazione di reimpostare il profilo di configurazione.

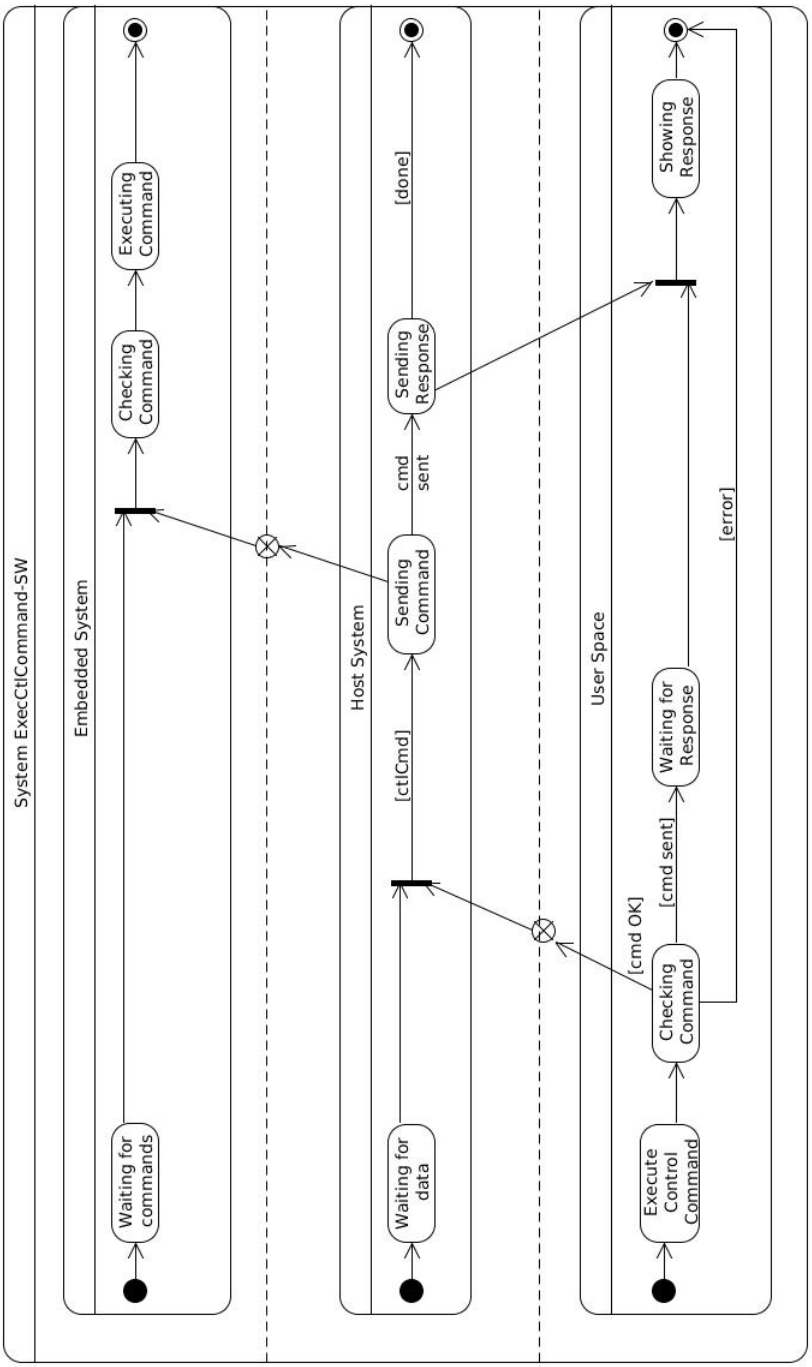

Figura 5.7: Diagramma degli stati che descrive l'esecuzione di comandi di controllo invocata dal lato host.

Inviare messaggi MIDI a dispositivi esterni

In questo caso (Figura 5.8), un ulteriore end-user entra a far parte del diagramma, ovvero il dispositivo MIDI esterno in attesa di messaggi.
Questo scenario si presenta nel momento in cui un'applicazione MIDI nello user space invia dei messaggi MIDI attraverso la porta di MIDI-OUT dello strumento (si ricorda che, perché tale porta sia istanziata presso ALSA, è necessario che il profilo di configurazione sia stato impostato con il MIDI-OUT attivato). Il sistema embedded, per aumentare l'efficienza e non ripetere i controlli, semplicemente si occupa di inoltrare il messaggio sulla sua porta di MIDI-OUT.

5.3 Struttura del sistema

Dopo aver dato l'analisi ad alto livello delle interazioni con gli end-user e dei flussi del sistema, in questa sezione verranno forniti i dettagli strutturali, ovvero le strategie architetturali con cui le varie parti del software saranno divise in "moduli" (non solo nel senso di moduli del kernel linux, ma anche in quello di "unità" di suddivisione delle funzionalità) e le relazioni che tali moduli avranno tra loro. Come dovrebbe essere chiaro a questo punto della trattazione, il sistema host dovrà fornire sia dei moduli del kernel che delle applicazioni nello user space che forniscono un'interfaccia all'interazione diretta con l'utente, quando questa è richiesta (ad esempio, per riconfigurare il profilo di configurazione o impostare la funzionalità di MIDI-OUT). Nel seguito verranno trattati i vari aspetti da considerare per l'implementazione delle componenti principali (come il profilo di configurazione e le applicazioni dello user-space) per poi (cfr. 5.3.6) definire la struttura generale del sistema software, sia host che embedded.

5.3.1 Il profilo di configurazione

Il profilo di configurazione dello strumento è senza dubbio una delle caratteristiche centrali per l'esecuzione del sistema. Come descritto nei requisiti `RSW_F.2` e `RSW_F.5` tale profilo è caricato dinamicamente al momento del collegamento fisico dello strumento all'host e, come descritto dal requisito `RSEC_NF.3`, esso viene fornito, almeno agli utenti con privilegi di amministrazione, come un file leggibile e modificabile all'interno del filesystem. Come riferito da numerose fonti però (tra cui, ad esempio, l'articolo in [50]) una "buona pratica" all'interno della programmazione del kernel dice che il codice del kernel non dovrebbe effettuare operazioni di I/O con file all'interno del filesystem, prima di tutto per evitare i classici errori di programmazione

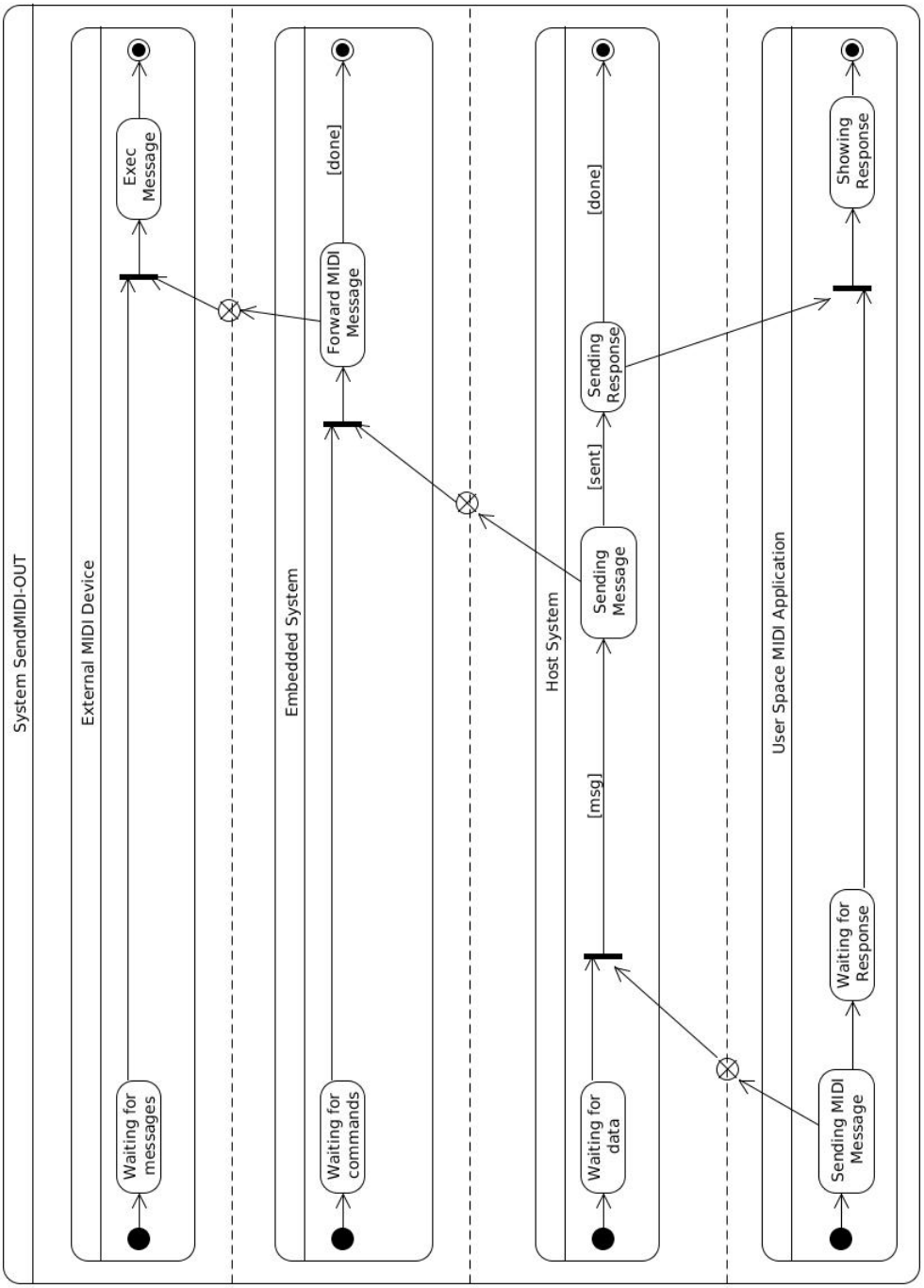

Figura 5.8: Diagramma degli stati che descrive l'invio di messaggi MIDI a dispositivi esterni.

che si effettuano nell'I/O, per garantire la portabilità del codice del kernel a prescindere dal filesystem e per non introdurre delle "politiche" all'interno del kernel.
Per evitarlo, dunque, si è utilizzato un approccio diverso. Come detto in precedenza (cfr. 3.3.3), il sysfs è un importantissimo strumento per il dialogo tra il livello applicativo e il kernel. Ogni dispositivo viene rappresentato con una directory all'interno di tale filesystem virtuale, contenente tutti gli attributi, sotto forma di file, che lo descrivono. Tali attributi, per i dispositivi USB, descrivono solitamente le caratteristiche strutturali del dispositivo, come il numero di endpoint o di interfacce. È chiaro, comunque, che ogni device driver ha la potenzialità di pubblicare tutte le informazioni che desidera. In particolare, implementare una gerarchia di directory e file all'interno del sysfs è abbastanza semplice: semplificando, ogni kobject viene rappresentato come una directory e ogni attributo come un file all'interno della directory, per cui è possibile pubblicare le informazioni in modo gerarchico a piacere con questi strumenti. Per il test di quanto affermato, è stata scritta una piccola funzione (come estensione del device driver *minisnd* presentato sopra, cfr. 3.5) che pubblica sotto la directory (kobject) del dispositivo un'altra directory (chiamata *ProvaConfig*) e un file all'interno di essa. La Figura 5.9 mostra il codice di questa funzione.
La struttura `thmx_config` si presenta come segue:

```
struct thmx_config {
  char *config_name;
  struct kobject *config_kobj;
  char *attr1;
};
```

`config_name` è il nome della directory base, rappresentata dal kobject `config_kobj`; la stringa `attr1` mantiene l'informazione della stringa da pubblicare all'interno dell'attributo. La funzione, dopo aver inizializzato il kobject, inizializza l'importante struttura `kobj_type`. Questa contiene la funzione `release` (che viene richiamata quando il *reference count* per questo kobject arriva a zero), una lista di attributi e un puntatore ad una struttura `sysfs_ops`. Quest'ultima contiene soltanto due funzioni: la `show()` viene richiamata ogni volta che si tenta di leggere i file degli attributi di questo tipo, e la `store()` implementa la scrittura su tali file. Per questo esempio, le funzioni che vengono salvate su questi puntatori sono solo fittizie, ma altrimenti possono implementare la lettura e scrittura dei dati sui file del sysfs. In particolare, la `show()` sarà implementata ritornando la stringa relativa all'attributo da leggere, mentre la `store()` modificherà l'attributo.
Dopo aver inizializzato le varie strutture, la creazione del kobject e dei suoi

```
/* create a dummy kobject (directory) within the device
 * kobject and a dummy file (attribute) within
 * this directory
 */
static int init_sysfiles(struct minisnd **dev) {
   struct kobj_type *ktype;
   struct attribute *attr;
   struct thmx_config *config;
   int retval;

   if (!dev || !(*dev))
      return -EINVAL;

   config = kmalloc(sizeof (*config), GFP_KERNEL);
   if (!config)
      return -ENOMEM;
   memset(config, 0, sizeof(*config));

   config->config_kobj = kmalloc(sizeof(struct kobject),
                                 GFP_KERNEL);
   if (!config->config_kobj)
      return -ENOMEM;
   memset(config->config_kobj, 0, sizeof(struct kobject));

   config->config_name = "ProvaConfig";
   config->attr1 = "Attributo1";

   attr = kzalloc(sizeof (struct attribute), GFP_KERNEL);
   if (!attr)
      return -ENOMEM;

   attr->name = config->attr1;
   attr->owner = THIS_MODULE;
   attr->mode = S_IRUGO;

   ktype = kzalloc (sizeof (struct kobj_type), GFP_KERNEL);
   if (!ktype)
      return -ENOMEM;

   ktype->release = kobj_release;
   ktype->sysfs_ops = &sysops;
   ktype->default_attrs = kzalloc(sizeof (struct attribute*),
                                  GFP_KERNEL);
   ktype->default_attrs[0] = attr;
```

```
    kobject_init(config->config_kobj, ktype);
   retval = kobject_add(config->config_kobj,
                        &((*dev)->dev->kobj),
                        config->config_name);
   if (retval < 0)
      return -EINVAL;

   (*dev)->config = config;
   return 0;
}
```

Figura 5.9: Funzione di esempio per la creazione di file e directory all'interno del sysfs

attributi viene effettuata dalle chiamate alle due funzioni `kobject_init()` e `kobject_add()`. In quest'ultima, è importante notare che il secondo argomento è un puntatore al kobject contenuto nel dispositivo USB creato: in questo modo, stiamo dicendo al kernel di creare una sotto-directory della directory relativa al dispositivo.
Dopo l'esecuzione del driver, è stata creata la directory */sys/class/usb/minisnd_ snd0*, al cui interno troviamo la directory *device/* e, all'interno di questa, la nostra *ProvaConfig/* che contiene il file Attributo1.

Struttura del profilo di configurazione

A questo punto possiamo definire il comportamento di inizializzazione del profilo di configurazione. In Figura 5.10 è presentato un *diagramma di sequenza*, che fornisce un facile modo di visualizzare le sequenze di eventi e verrà ampiamente usato in seguito (cfr. 5.4), con il flusso di azioni per la creazione del file del profilo di configurazione.
Sostanzialmente verrà implementato uno script o una piccola applicazione nello user space. Questa verrà attivata da un'opportuna regola *udev* collegata alla creazione dell'attributo corrispondente all'ultimo campo del file di configurazione: in questo modo, quando l'applicazione sarà eseguita sarà certa che la rappresentazione del profilo di configurazione all'interno del sysfs è completata. A questo punto, lo script leggerà uno dopo l'altro gli attributi e scriverà un file *human readable*, leggibile e modificabile dal solo amministratore. Per evitare che un utente senza privilegi possa tentare di leggere/scrivere direttamente i file del sysfs, anche questi verranno creati con privilegi relativi al solo amministratore.

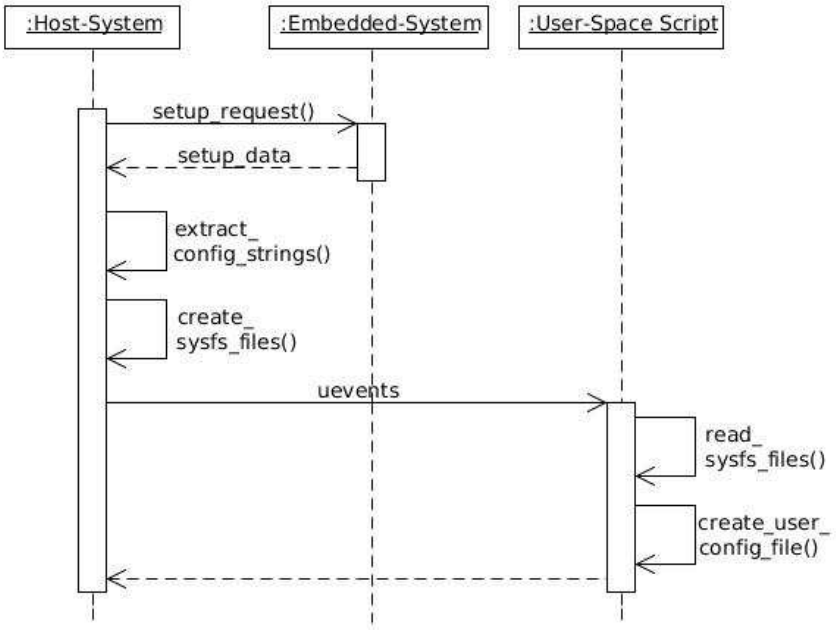

Figura 5.10: Diagramma della sequenza relativa all'inizializzazione del profilo di configurazione.

Di seguito viene riportata nel dettaglio la gerarchia che verrà creata all'interno del sysfs per rappresentare il profilo di configurazione.

```
ConfigurationProfile
 -> poliphony      (A: bool)
 -> numVoices      (A: int)
 -> playingMode    (A: string)
 -> midiOUT        (A: bool)
 -> programmableHW (K)
     -> hw-x       (A: string)
     [->...]
 -> controlCmd     (K)
     -> cmd-x      (A: string)
     [->...]
 -> associations   (K)
     -> as-x       (A: string)
     [->...]
```

Per ogni elemento, è stato specificato accanto se si tratta di un kobject (`K`) e, quindi, apparirà come una directory, oppure di un attributo (`A`), per cui apparirà come un file. Inoltre, per gli attributi, è specificato quale sarà la tipologia di dati che conterrà. Per quanto riguarda le ultime tre directory, ogni attributo in esse contenuto sarà nella forma `oggetto-x`, dove `x` è un indice incrementale. Il contenuto di questi file sarà una stringa, la cui semantica varia a seconda della directory: gli attributi sotto la directory `programmableHW` saranno stringhe con la descrizione del tipo di hardware disponibile e l'indice, separati dal carattere ":". Per il prototipo del Thereminux è stata inclusa la sola stringa `PUSHBUTTON:0` visto che, come detto in precedenza (cfr. 4.1), è presente un unico pulsante programmabile a rappresentare questi elementi. Per quanto riguarda i comandi di controllo, questi sono stati scritti sui file tramite stringhe nella forma `COMMAND:x`, con `x` l'indice del comando. Infine, i file con le associazioni contengono delle stringhe nella forma "`x:y`", dove `x` e `y` sono valori interi rappresentanti, rispettivamente, un indice di un elemento hardware e un indice di un comando.
Vista l'organizzazione con cui il profilo di configurazione viene impostato nel sysfs, segue che anche sul sistema embedded i dati relativi al profilo di configurazione sono stati mantenuti in diversi descrittori stringa, ognuno formattato come `nome_attributo=valore_attributo`, dove il nome dell'attributo è quello che diventerà il nome del file relativo sul sysfs.
Il file prodotto dallo script in user space è stato posto in una directory di configurazione del sistema (in questa fase prototipale del sistema, all'interno del Desktop) e contiene i vari elementi in uno stile simile all'XML: il file di default, così come appare formattato, è quello che segue:

```
<ConfigurationProfile>
```

```
    <poliphony>false</poliphony>
    <numVoices>1</numVoices>
    <playingMode>glissando</playingMode>
    <midiOUT>true</midiOUT>
    <programmableHW>
       <hw-0>PUSHBUTTON:0</hw-0>
    </programmableHW>
    <controlCmd>
       <cmd-0>RESET:0</cmd-0>
    </controlCmd>
    <associations>
       <as-0>0:0</as-0>
    </associations>
 </ConfigurationProfile>
```

Per quanto riguarda, infine, il controllo in fase di reimpostazione del profilo, l'applicazione che viene chiamata dall'utente amministratore si preoccupa di effettuare il controllo sintattico sul file del profilo, mentre successivamente si occupa di scrivere sui file del sysfs per invocarne la modifica. È al metodo `store()` all'interno del driver che spetta, a quel punto, di effettuare un controllo sui valori immessi, in modo da evitare che siano salvate stringhe che non hanno significato sul dispositivo (ad esempio hardware e/o comandi inesistenti).
A questo punto, è possibile che si crei il seguente scenario: un utente con privilegi di amministrazione potrebbe scrivere direttamente sui file del sysfs, modificandone il valore correttamente, ma non passando per il comando di conferma del profilo di configurazione. In questo caso, il profilo rimarrebbe in uno stato obsoleto rispetto al reale valore mantenuto dal driver. Per ovviare a ciò, sarebbe stato necessario che ogni chiamata alla funzione `store()` causasse un controllo del file del profilo di configurazione per verificarne l'integrità rispetto ai dati contenuti nel kernel e sul gadget. Il problema in questa soluzione sta nell'evidente inutilità di effettuare questo controllo anche quando è il file di configurazione stesso ad essere stato modificato e le chiamate alla `store()` si effettuano come una registrazione di queste modifiche tramite l'apposito script di conferma. Infatti, qui non è possibile che i dati inviati al kernel siano diversi da quelli del file di configurazione, derivando questi dati proprio da tale file.
Per questo, è stata utilizzata una doppia strategia per far sì che, nel caso in cui è l'apposito programma di riconfigurazione del profilo a scrivere sui file del sysfs, il controllo di integrità del file di configurazione non venga fatto, ma invece venga effettuato solo quando un utente (root) esegue delle scritture

direttamente sui file del sysfs:

- il programma di riconfigurazione del profilo aggiunge ai buffer di scrittura sul sysfs una stringa di prefisso (la stringa `THMXCFG_`) che, quando riconosciuta dal kernel, previene l'invocazione del controllo sul file di configurazione;

- se questa stringa di prefisso non è presente, il driver crea un `uevent` che, attraverso un'opportuna regola di udev, fa sì che venga eseguito uno script che controlla lo stato di aggiornamento del file di configurazione e, se non dovesse risultare aggiornato, richiama lo script di inizializzazione del file, che ne ricostruisce i valori. L'evento di udev che viene inviato crea una variabile di ambiente in cui viene pubblicato il nome del campo modificato e il suo valore attuale. Lo script, in questo modo, potrà verificare se il campo salvato sul file risulta aggiornato. La variabile di ambiente è così formata:

  ```
  THMX_CONFIG=MODIFY:name==value
  ```

  dove `name` è il nome del campo del profilo modificato e `value` è il suo valore.

La limitazione che salta subito all'occhio per quanto riguarda questa strategia è che, poiché verosimilmente l'utente privilegiato può conoscere la stringa di prefisso, potrebbe usarla lui stesso nelle scritture a mano sul sysfs, bypassando così il controllo e facendo sì che gli uevent non siano creati opportunamente. In ogni caso, la soluzione è sembrata comunque ragionevole perché tali scritture sono permesse solo all'utente privilegiato, per cui sembra opportuno assumere che le sue azioni siano affidabili.

5.3.2 Script e applicazioni in *user space*

Verranno qui fornite le specifiche dei vari script (o applicazioni se troppo complesse per essere implementate come semplici script) che sono stati implementati nello user space del sistema host al fine di creare un'interfaccia con gli utenti o, semplicemente, delle funzioni batch come quelle descritte nel paragrafo precedente.

- `thmx_init_profile`: script batch per l'inizializzazione del file del profilo di configurazione a partire dai file del sysfs (cfr. 5.3.1);

- `thmx_check_profile`: script batch che controlla lo stato di aggiornamento dei file del profilo di configurazione. Per farlo va a leggere la variabile di ambiente `THMX_CONFIG` generata dall'*uevent* e, da questa, ricava il nome dell'ultimo campo modificato e il suo valore attuale, così da poterlo confrontare con il valore scritto sul file di configurazione per quel campo;

- `thmx_confirm_profile`: script che può eseguire soltanto l'utente amministratore e che fa sì che un nuovo profilo di configurazione venga impostato. Lo script esegue un controllo sulla sintassi del file di configurazione e, se corretta, invia i dati al kernel tramite scritture sui file del sysfs, applicando a queste scritture la stringa di prefisso che fa sì che non vengano generati *uevent*;

- `thmx_set_midi_out <true/false>`: script eseguibile da un utente qualsiasi. Imposta la funzionalità di MIDI-OUT attraverso lo strumento. Semplicemente effettua una scrittura sul file *ConfigurationProfile/midiOUT* all'interno del sysfs. Questo può provocare l'aggiornamento del file di configurazione (tramite la generazione dell'*uevent*) e l'invio del nuovo valore al gadget se il valore dell'attributo cambia;

- `thmx_read_profile`: script eseguibile da un utente qualsiasi. Semplicemente legge il file del profilo di configurazione e lo stampa formattato;

- `thmx_read_ctl_commands`: script eseguibile da un utente qualsiasi. Questo legge il profilo di configurazione e stampa i comandi di controllo in esso contenuti;

- `thmx_exec_ctl_command <nome_comando>`: script eseguibile da un utente qualsiasi. Permette di eseguire un comando di controllo sul dispositivo hardware. Questi comandi, dopo aver superato il check da parte dello script stesso che controlla se esistono tra quelli disponibili, verranno inviati al kernel tramite funzioni `ioctl()`. Per la lista dei comandi che sono stati implementati si veda 5.3.4;

- `thmx_edit_profile`: script eseguibile soltanto dall'utente privilegiato. Semplicemente apre il file di configurazione utilizzando l'editor *vim*;

- `thmx_system_clean`: script batch invocato da una regola udev eseguita nel momento della disconnessione dello strumento. Esso si occupa di scollegare i moduli del driver, eliminare i nodi sotto */dev* e il file di configurazione creato;

- `thmx_debug_get_audio`: script di debug eseguibile soltanto dall'utente privilegiato. Esso è stato sviluppato, come verrà trattato in seguito (cfr. 7) per ottenere una registrazione, in tempo reale, dell'audio che viene inviato al sistema tramite il Thereminux. Tale stream audio viene salvato su un file audio grezzo che è possibile ascoltare con opportune applicazioni audio standard;

- `thmx_config.rules`: file installato sotto la cartella */etc/udev/rules.d* che contiene le regole udev per l'esecuzione del sistema come descritto finora;

- `thmx_node_create`: programma in esecuzione sul sistema embedded che viene eseguito tra gli script di avvio del sistema operativo e crea il nodo sotto */dev* che rappresenta il dispositivo a caratteri implementato, tra le altre interfacce esposte, dal modulo installato sul sistema embedded (cfr. 5.3.6);

- `thmx_gadget_alsa_capture`: programma in esecuzione sul sistema embedded che legge i campioni audio in ingresso e li invia al device driver gadget utilizzando la sua interfaccia a caratteri (per i dettagli cfr. 5.3.6);

- `thmx_alsa_capture_start`: script eseguito tra quelli di avvio del sistema operativo del sistema embedded e il cui compito è semplicemente di avviare l'esecuzione del programma `thmx_gadget_alsa_capture`.

Oltre a questi, sono stati implementati alcuni script di utilità che hanno agevolato il lavoro di sviluppo del sistema:

- `cp_thmx_mod_fs`: script usato per il sistema host, con il quale il modulo compilato viene copiato in una sotto-directory della directory di sistema adibita ai moduli (*/lib/modules/*) e viene invocato il comando `depmod`;

- `cp_thmx_mod_sd`: script usato per il sistema embedded, con il quale il modulo compilato viene copiato sulla scheda SD che contiene il file system usato sulla Beagle Board. Anche in questo caso, il modulo viene copiato in una sotto-directory di */lib/modules*;

- `beagle-setup-sd`: script eseguito per effettuare il completo setup della scheda SD usata sulla Beagle Board. Questo script effettua la formattazione della scheda, la copia dei file relativi al bootloader della Beagle Board e del sistema operativo e, infine, la copia dei file relativi al Thereminux (modulo e script a livello applicativo).

5.3.3 Endpoint presenti sul dispositivo

Date le funzionalità implementate, gli endpoint che vengono forniti dal sistema embedded del Thereminux sono quattro (si ricorda che si definisce di *input* un endpoint che invia dati dal dispositivo all'host e di *output* il viceversa, cfr. 3.2):

1. uno di input di tipo *bulk* per la gestione di alcune informazioni che il dispositivo invia all'host, come stringhe di debug o di controllo dello stato di esecuzione;
2. uno di input di tipo *isochronous* per la gestione del flusso di campioni audio che dovranno essere trasformati in MIDI;
3. uno di output di tipo *interrupt* per la gestione dei dati, come i comandi di controllo inviati da software, che devono essere gestiti nel più breve tempo possibile;
4. un altro di output di tipo *interrupt* specifico per l'invio di messaggi MIDI di output.

5.3.4 Comandi ioctl implementati

I comandi `ioctl` che forniti sono:

- `ISO_IN_ZERO`: azzera i valori delle stringhe che vengono inviate dal dispositivo tramite l'endpoint isochronous di input;
- `ISO_IN_MULT <val>`: moltiplica i valori binari che verranno inviati sull'endpoint isochronous di input per la costante moltiplicativa `val`;
- `BULK_IN_DBG`: fa sì che il dispositivo invii una stringa di debug sull'endpoint bulk di input;
- `ISO_IN_DBG`: fa sì che il dispositivo invii una stringa di debug sull'endpoint isochronous di input;

5.3.5 Regole udev

Sono state implementate le seguenti regole udev presenti nel file `thmx_config.rules`:

- caricamento/liberazione dei moduli in plug-n-play;
- esecuzione dello script di inizializzazione del profilo di configurazione (`thmx_init_profile`);

- esecuzione dello script per il controllo dell'aggiornamento del profilo di configurazione (`thmx_check_profile`).
- esecuzione dello script per la pulizia del sistema (`thmx_system_clean`).

5.3.6 Il diagramma dei moduli

Le informazioni ricavate dalle analisi precedenti hanno permesso, infine, di sviluppare una modularizzazione delle funzionalità da inserire nel sistema host così come in quello embedded. Prima di entrare nel dettaglio di ognuno dei due, qualche considerazione generale. Si è utilizzato un modello di progettazione "object-like", nel senso che, come risultato dell'analisi delle funzionalità necessarie, il codice è stato diviso in diversi moduli e file. In particolare, ogni file implementa una funzionalità ben precisa ed è costituito, in maniera simile agli oggetti del paradigma object-oriented, da una serie di attributi propri dell'oggetto e da alcune funzioni per lavorare su tali attributi. Per evitare confusione, si parlerà qui di "moduli" quando ci si riferisce ai moduli del kernel, mentre saranno definit "oggetti" i file che compongono i vari moduli.
Gli attributi di ogni oggetto sono stati inseriti in una apposita `struct`, così che esso risulti chiaramente incapsulato. Inoltre, ogni oggetto ha delle funzioni "pubbliche", ovvero che altri oggetti possono richiamare, e altre che sono definite "private", cioè per uso interno dell'oggetto. Queste ultime sono state dichiarate, seguendo la sintassi C, come `static`.
Passiamo ora ad esaminare nel dettaglio i due sistemi, host ed embedded.

Moduli e oggetti per il sistema host

Il diagramma che rappresenta la struttura degli oggetti e dei moduli del sistema host è mostrato in Figura 5.11.
Come è possibile vedere, è stato previsto lo sviluppo di tre diversi moduli del kernel, denominati `usb_module`, `snd_module` e `vtm_module`. Di questi, il primo risulta quello centrale, che implementa effettivamente il device driver USB per il dispositivo. Gli altri due sono stati pensati in modo tale da rendere totalmente indipendente il livello di gestione del dispositivo USB e fornire delle funzionalità generiche e pienamente "riusabili" su altri dispositivi che rispettano l'interfaccia del Thereminux. Tali funzionalità sono la creazione e la gestione di un dispositivo ALSA costruito su un dispositivo esistente di "tipo" Thereminux e la conversione Voltage-To-MIDI di buffer di campioni audio. In questo modo si è cercato di seguire quanto affermato nella presentazione generale del lavoro (cfr. 1.1), e cioè che il sistema software avrebbe

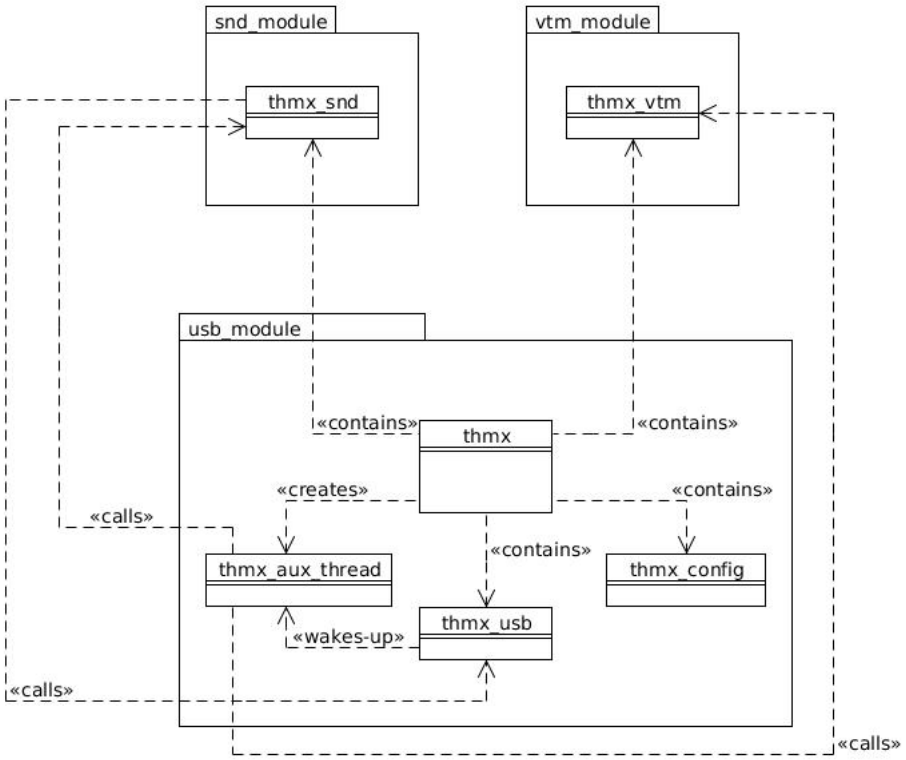

Figura 5.11: Diagramma dei moduli e degli oggetti relativi al sistema host

implementato "un interprete di segnali di voltaggio che arrivano da uno strumento analogico, elettronico o elettrificato, generico".
Si può parlare di una sorta di "tipo" Thereminux in quanto tutti i moduli condividono una singola struttura di dati chiamata, appunto, `thmx`. In altre parole, è stato pensato un sistema di comunicazione basato sull'incapsulamento delle informazioni attraverso strutture di contenimento dei dati condivise. I dettagli di come ciò funzioni saranno chiari man mano che si entrerà più in profondità nella struttura del sistema. I moduli di "utilità" (`snd_module` e `vtm_module`) esportano le loro funzioni pubbliche come simboli del kernel (utilizzando la `EXPORT_SYMBOL()`) in modo tale che il modulo principale possa richiamarle. Il modulo principale è, come detto, `usb_module`. Questo definisce tutti gli oggetti necessari per la gestione dei dati del sistema: c'è un oggetto che contiene i dati per la gestione della comunicazioni USB (`thmx_usb`), un oggetto per la gestione dei dati relativi al profilo di configurazione, ecc. Tutti i dati definiti da tali oggetti vengono mantenuti all'interno di un'unica struttura (chiamata `struct thmx`). Tale struttura viene passata alle funzioni proprie dei moduli di utilità, che la potranno modificare e/o leggere. Per capire alcuni aspetti di come questo viene effettuato, vediamo nel dettaglio i vari oggetti del modulo `usb_module`:

- `thmx_usb` contiene una struttura con le informazioni sugli endpoint del sistema e le funzioni per operare su di essi. Come vedremo nella descrizione dell'implementazione (cfr. 6), inoltre, è stata definita all'interno di questo oggetto una struttura di tipo `thmx_usb_ops`. Questa, in realtà, contiene un solo puntatore a funzione (la *write*) che viene utilizzata per la comunicazione tra il modulo `snd_module` e quello `usb_module`. Infatti la creazione del dispositivo MIDI fa sì che ALSA crei i suoi propri nodi nel filesystem (solitamente sotto */dev/snd/*). Le applicazioni ALSA-compliant, come jack, prendono come riferimento questi nodi e non quelli creati dal modulo USB. In questo modo non si avrebbe accesso all'implementazione dei metodi per la reale lettura/scrittura di dati USB dal modulo `snd_module`, visto che questi vengono dichiarati dentro `usb_module`. Al fine di risolvere questa questione, per effettuare la scrittura di dati MIDI sul dispositivo USB quando questi vengono inviati al Thereminux usando il file ALSA, si salverà un puntatore alla funzione che si occupa di inoltrare questi dati MIDI al sistema embedded, funzione di cui è mantenuto un riferimento dentro la struttura `thmx_usb`, che è contenuta a sua volta dentro `thmx`. Un puntatore alla `struct thmx` verrà sempre salvato tra i dati privati delle altre strutture di utilità (ad esempio, la `struct snd_card` di ALSA), per cui sarà sempre accessibile dai diversi moduli e dalle diverse funzioni di call-

back. In questo modo, quando il `snd_module` riceve dati sul file ALSA, può inoltrarli al modulo `usb_module` utilizzando la funzione salvata. Questa è la motivazione della relazione `«calls»` descritta nel diagramma in Figura 5.11.
Oltre a questo, è interessante notare la presenza di una relazione tra `thmx_usb` e l'oggetto `thmx_aux_thread`. Quest'ultimo, come verrà spiegato tra poco, implementa un kernel thread per la gestione dell'acquisizione di campioni audio e la loro conversione in MIDI. Tale thread rimane bloccato, però, finché non ci sono dati in ingresso, e viene "avvertito" della presenza di questi dati in ingresso dal modulo `thmx_usb`. In altre parole, quando viene completata una comunicazione USB e risultano pronti dei dati in ingresso all'host, la callback eseguita al termine della transazione USB non fa altro che salvare il contenuto del buffer di input all'interno della struttura `thmx` e svegliare il thread, passandogli la struttura. I dettagli di come questo flusso avviene verranno dati nel Paragrafo 5.4 e nel Capitolo 6;

- `thmx_aux_thread` esegue, come detto nel punto precedente, un thread di utilità che si occupa di invocare la conversione Voltage-To-Midi passandogli il buffer che è stato aggiornato da `thmx_usb` e invare il risultato al livello applicativo utilizzando le funzioni ALSA messe a disposizione in `thmx_snd` (si noti la relazione `«calls»` tra il `thmx_aux_thread` e `thmx_snd`;

- `thmx_config`: gestisce il profilo di configurazione, secondo quanto descritto in precedenza (cfr. 5.3.1). Quindi si occupa di creare i file del sysfs e implementare le funzioni per la lettura e la scrittura delle informazioni su tali file, nonché creare gli eventi di udev per l'esecuzione delle applicazioni nello user level per la gestione del file di configurazione nel file system;

- `thmx` è l'oggetto principale del modulo. Contiene un riferimento a tutte le strutture relative agli oggetti precedenti, e funziona da "coordinatore" di questi. Ad esempio, alloca tutte le strutture dati dei vari oggetti e, in fase di inizializzazione, richiama le loro funzioni di inizializzazione, registrando di fatto il dispositivo presso i vari sotto-sistemi (USB, ALSA). Inoltre, gestisce i metodi standard che vengono invocati dal livello applicativo sul nodo del dispositivo USB registrato. In particolare, questo dispositivo implementa soltanto le system call `open()` e `ioctl()` utili per l'I/O, visto che tutte le altre vengono effettuate indirettamente (tramite il file ALSA o scrivendo sul sysfs).

Moduli e oggetti del sistema embedded

Il sistema embedded (Figura 5.12), sebbene più semplice, è organizzato secondo principi simili all'host.

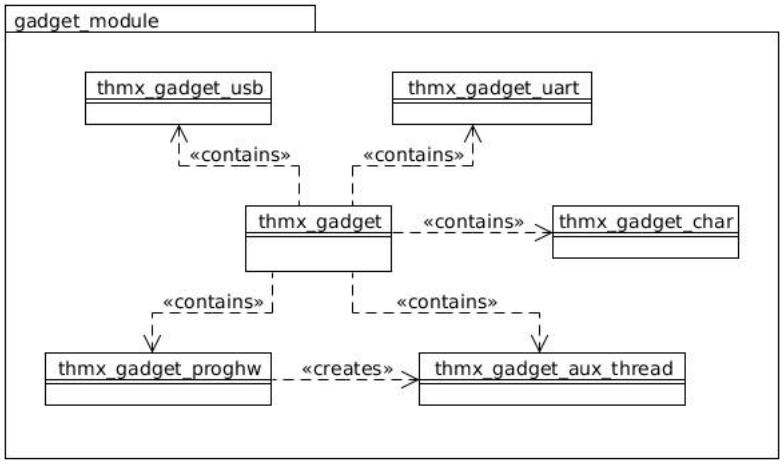

Figura 5.12: Diagramma dei moduli e degli oggetti relativi al sistema embedded

Come si nota, è costituito da un unico modulo del kernel che ne incorpora le varie funzionalità. Allo stesso modo del sistema host, anche qui troviamo un oggetto principale che funge da contenitore delle strutture relative agli altri oggetti e ne richiama le funzioni. Un'analisi più in dettaglio di queste funzioni permetterà di comprendere anche il funzionamento di base del dispositivo, per cui vediamole singolarmente:

- `thmx_gadget_usb` è l'oggetto che definisce le strutture e le funzioni per l'implementazione della comunicazione USB con l'host. Contiene una struttura (`thmx_gadget_config`) con cui definisce il profilo di configurazione come un insieme di descrittori stringa, definisce gli endpoint e tutte le strutture necessarie per implementare un dispositivo USB gadget (cfr. 3.7.1);

- `thmx_gadget_uart` implementa un driver seriale basato sul modello *TTY* per la comunicazione con la porta RS-232 della Beagle Board. Per questo, utilizza l'API *UART* definita nel file *include/linux/serial_ core.h*. Tramite questa, il driver implementa un dispositivo seriale per la comunicazione di dati MIDI di output verso dispositivi esterni;

- `thmx_gadet_char`: per l'acquisizione dei campioni audio in ingresso, è sembrato inutile implementare un driver ALSA che ricalchi interamen-

te quello già esistente all'interno del kernel. Per questo, si è deciso di utilizzare un'applicazione ALSA che gira nello user space e implementa un loop di letture sul dispositivo audio della Beagle Board, che è accessibile tramite il driver ALSA standard. Ogni volta che i dati sono disponibili, questa applicazione riempie un buffer e lo manda al modulo `gadget_module` tramite normali system call come la `write()`. Il problema, però, è che l'interfaccia USB Gadget non mette a disposizione le system call per l'I/O dal livello applicativo. Per questo il driver implementa un semplice dispositivo a caratteri sul quale il livello applicativo può inviare le sue `write()`. L'implementazione di questa system call semplicemente inoltra i dati sull'endpoint isochronous di input all'host;

- `thmx_gadget_proghw` è un oggetto che implementa un elemento hardware programmabile sul dispositivo. Questo significa semplicemente che viene salvato un riferimento alla linea di irq a cui è associato quell'hardware, in modo da registrare un interrupt handler per tale irq ed eseguirlo ogni volta che un interrupt arriva su quella linea. Questo è essenzialmente ciò che questo oggetto fa. L'unica cosa su cui c'è da prestare attenzione è che l'esecuzione del comando di controllo associato all'hardware programmabile andrebbe effettuata all'interno dell'interrupt handler, ma questo è rischioso perché ci si trova in un contesto in cui non si può andare in sleep o prendere lock, e il comando di controllo da eseguire potrebbe, in linea di principio, effettuare operazioni arbitrarie. Per questo, l'handler crea un kernel thread che si occupa di eseguire il comando corrente associato all'hardware programmabile che ha inviato l'interrupt. In questo modo, l'interrupt handler rimane il più efficiente possibile. Inoltre, il fatto che l'hardware programmabile sulla Beagle Board sia, in realtà, un pulsante hardware ha fatto sì che, per la gestione di tale dispositivo di input, è stato necessario implementare un piccolo device driver *GPIO* (General Purpose Input/Output). Questo tipo di dispositivi (si veda il file *Documentation/gpio.txt* all'interno dei sorgenti del kernel) fornisce dei semplici segnali controllati da software e, quindi, può essere associato a diversi elementi fisici trovati sulle comuni schede hardware (pulsanti, linee JTAG, ecc). Nel nostro caso, è stato necessario registrare il driver GPIO per il pulsante programmabile, ricavando da questo la linea di irq associata a tale elemento hardware, per poi usarlo nell'applicazione;

- `thmx_gadget_aux_thread`: come spiegato al punto precedente, questo è un kernel thread che si occupa di eseguire una singola funzione associata all'hardware programmabile che ha inviato un interrupt.

La struttura presentata in questa sezione verrà approfondita ulteriormente quando si tratteranno nel dettaglio gli aspetti di implementazione del sistema (cfr. Capitolo 6). Per concludere la descrizione progettuale, verrà fornita di seguito l'analisi delle sequenze di esecuzione principali del Thereminux (cfr. 5.4).

5.4 Analisi delle sequenze

Sequenze del sistema host

Iniziamo con la descrizione dei flussi operativi dal punto di vista del sistema host. Per la spiegazione che segue, sono stati presi in considerazione tutti i casi d'uso descritti in precedenza (cfr. 5.1) più altre operazioni fondamentali, come l'inizializzazione del sistema. La Figura 5.13 mostra, appunto, proprio il flusso relativo all'inizializzazione del sistema host.
Per evidenziare il contributo dei vari moduli del kernel che fanno parte del sistema host, gli oggetti sono stati inseriti all'interno di riquadri che portano il nome dei moduli stessi. La sequenza di inizializzazione corrisponde, dal punto di vista dell'host, alla funzione `probe()` implementata all'interno dell'oggetto `thmx`. Come si vede in Figura, questa non fa altro che allocare le varie strutture dati inizializzandone i parametri di base, per poi richiamare una per una le funzioni di inizializzazione degli altri oggetti e moduli, le quali effettueranno le proprie allocazioni e impostazioni e salveranno il tutto nella struttura passata loro come argomento, la `struct thmx`, appunto. C'è da notare che i diagrammi mostrati non pretendono di essere completamente esaustivi, ma spesso è possibile incontrare delle "funzioni" che non sono realmente presenti nell'API, ma descrivono un certo comportamento che, seppur importante all'interno della sequenza, è ad un livello di dettaglio troppo particolareggiato per essere incluso punto per punto nel diagramma, pena un drastico peggioramento della leggibilità. Si è preferito dunque mostrare il flusso principale e tralasciare i dettagli eccessivi, che sono ovviamente forniti con il codice dell'implementazione.
In Figura 5.13 c'è da notare la creazione di due kernel thread: infatti, uno è rappresentato dal `thmx_aux_thread` già trattato: questo, in fase di inizializzazione, non fa altro che bloccarsi su una coda di attesa per essere risvegliato solo quando sono presenti dati di input da gestire. Oltre a questo, l'oggetto `thmx_usb` crea un altro kernel thread che non è stato mostrato nel diagramma di Figura 5.11 (cfr. 5.3) in quanto interno all'oggetto `thmx_usb`: questo si occupa della ricezione vera e propria dei dati isochronous, eseguendo un loop di ricezioni continue sull'endpoint e gestendo i possibili errori. In particolare,

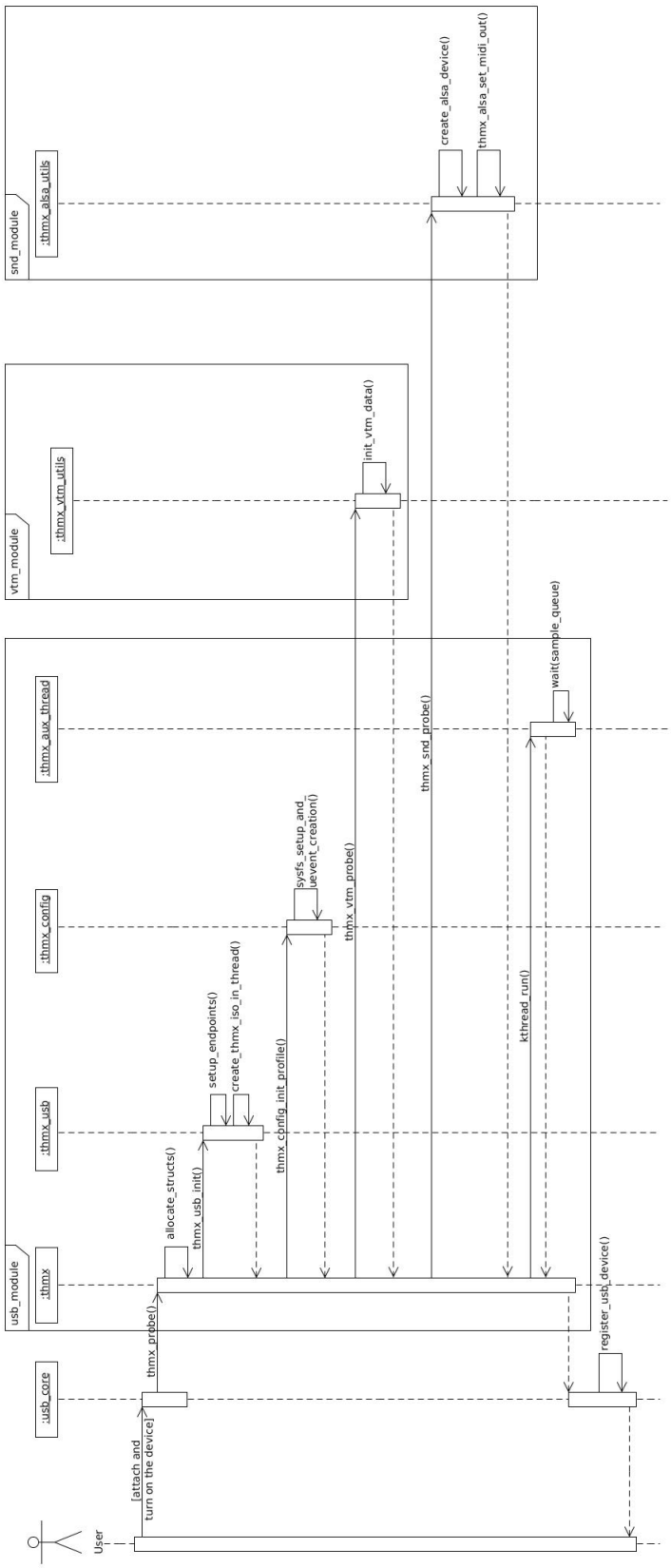

Figura 5.13: Diagramma della sequenza di inizializzazione del sistema host

questo thread si occupa di risvegliare l'aux thread quando ci sono dati disponibili, ma anche di verificare le ricezioni, implementando dei meccanismi di *recovery* quando, per qualche motivo, ci sono errori nella ricezione.
Lo scenario relativo all'esecuzione sullo strumento è, appunto, presentato in Figura 5.14.
La prima cosa che viene eseguita, quando viene completato l'URB relativo all'endpoint isochronous di input, è l'handler `thmx_read_iso_callback` dentro `thmx_usb`. Questo non fa altro che copiare il buffer di input e svegliare il thread, il quale richiama la funzione di conversione Voltage-To-Midi e, quando questa gli restitusce il messaggio MIDI contenente la nota MIDI risultata dalla conversione, lo invia al livello applicativo. Quest'ultima operazione è eseguita sfruttando l'API ALSA standard e, quindi, inviando i messaggi MIDI alle applicazioni che già sono collegate al dispositivo e sono in attesa di tali messaggi.
Andando avanti, la Figura 5.15 mostra il flusso di reimpostazione del profilo di configurazione una volta che l'amministratore lo ha modificato.
Come detto in precedenza (cfr. 5.3.1) tale operazione è stata implementata attraverso una serie di scritture sui file-attributo del dispositivo all'interno del sysfs, cosa che causerà l'invocazione di altrettante funzioni `store()`. Per tutte le modifiche che non riguardano né le associazioni tra elementi hardware programmabili e comandi di controllo, né la reimpostazione del MIDI-out, questa operazione lavorerà in locale, semplicemente aggiornando le variabili interne del driver e poi creando delle variabili di ambiente di udev per lo script `thmx_check_profile` (se nelle `store()` non è stato inviata la stringa di prefisso insieme ai valori dei campi del profilo), che controllerà l'aggiornamento del file del profilo. Quando invece le modifiche verranno apportate alla sezione `associations` o a quella `midiOUT` del profilo di configurazione, sarà necessaria una comunicazione con il dispositivo, che verrà effettuata richiamando un opportuno metodo di `thmx_usb`.
Ancora, la Figura 5.16 mostra la sequenza relativa all'esecuzione di comandi di controllo da software.
Tale sequenza è in realtà molto semplice: il programma `thmx_execCtlCommand`, infatti, non fa altro che eseguire delle `ioctl()` sul nodo del dispositivo USB. Questa system call verrà implementata semplicemente come un inoltro del comando, già precedentemente controllato dall'applicazione nello user space, al dispositivo remoto.
Infine, l'ultima operazione che è sembrato importante riportare nel dettaglio è l'invio di note MIDI di output. Questa sequenza è riportata in Figura 5.17. In questa sequenza è evidenziato il comportamento che è stato trattato nel paragrafo 5.3.6: un'applicazione ALSA, infatti, proverà a scrivere sul dispositivo ALSA craeto dal `snd_module`, ed il metodo standard per

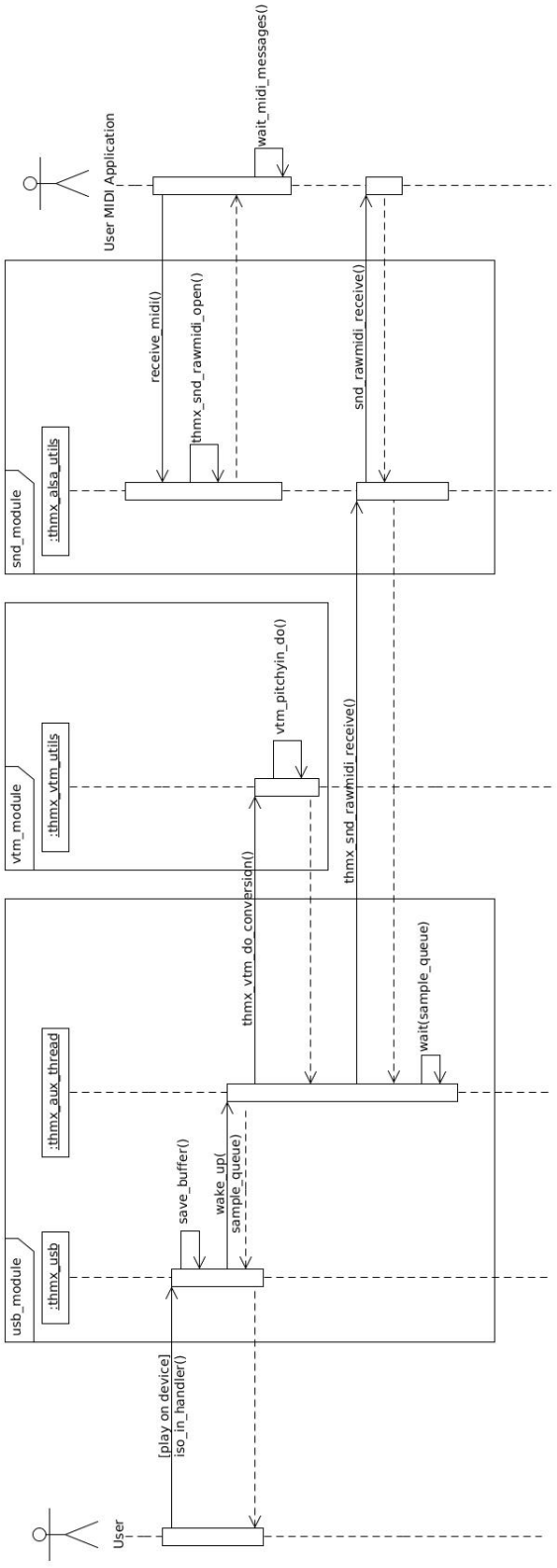

Figura 5.14: Diagramma della sequenza relativa all'arrivo di un buffer di campioni audio dall'USB

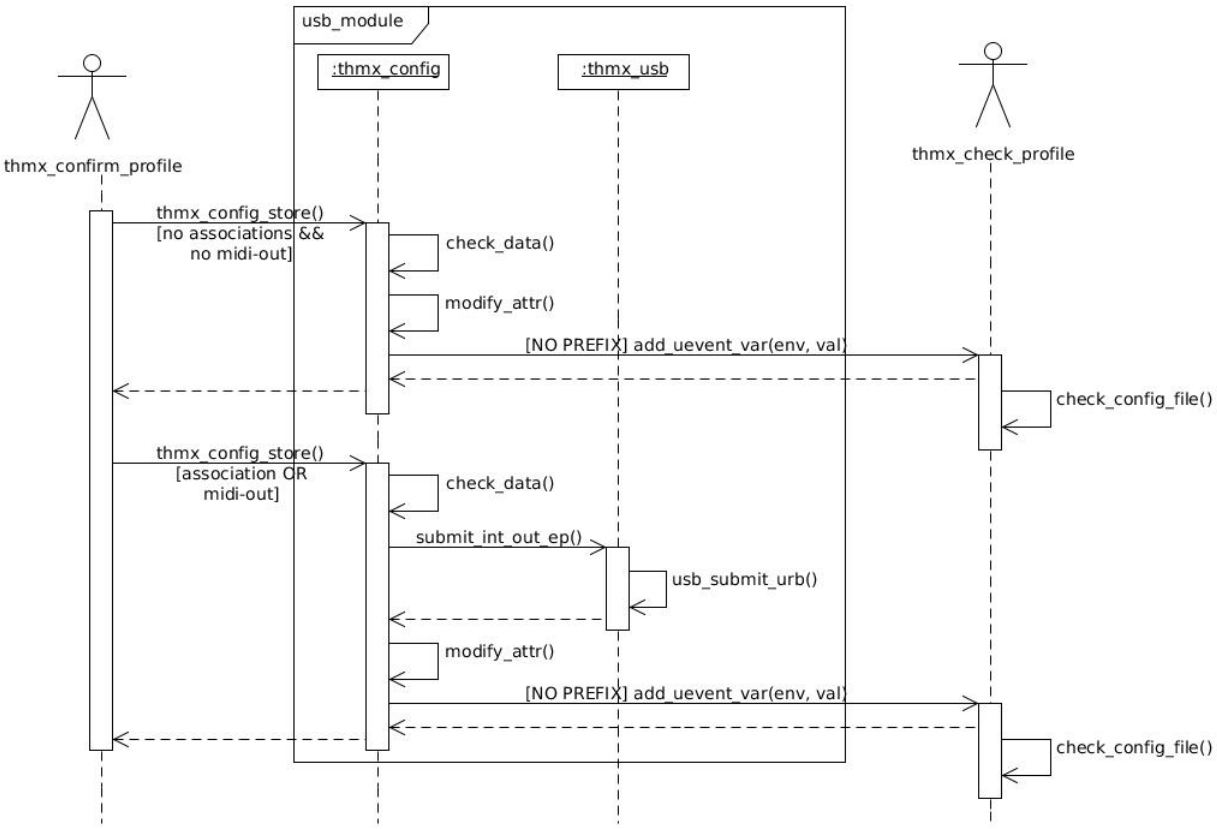

Figura 5.15: Diagramma della sequenza relativa alla reimpostazione del profilo di configurazione

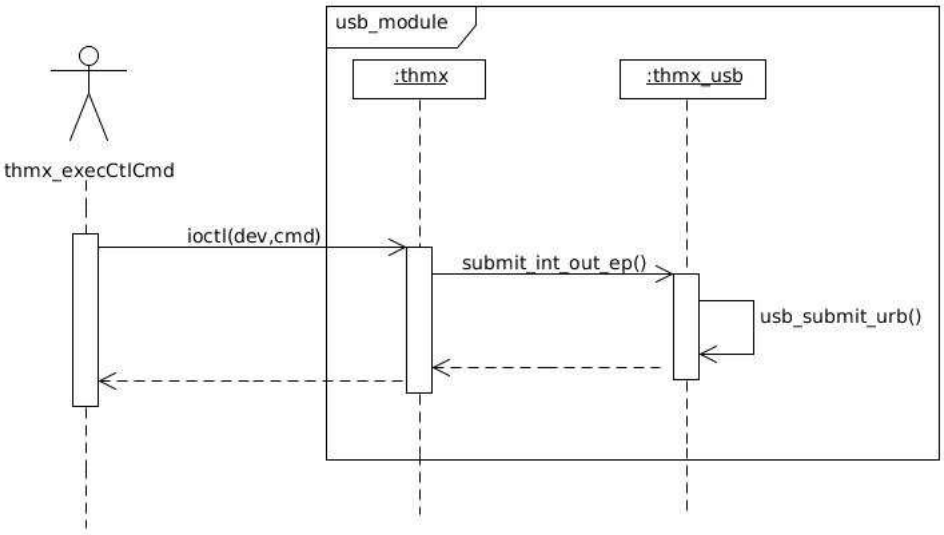

Figura 5.16: Diagramma della sequenza relativa all'esecuzione di comandi di controllo inviati dal sistema host

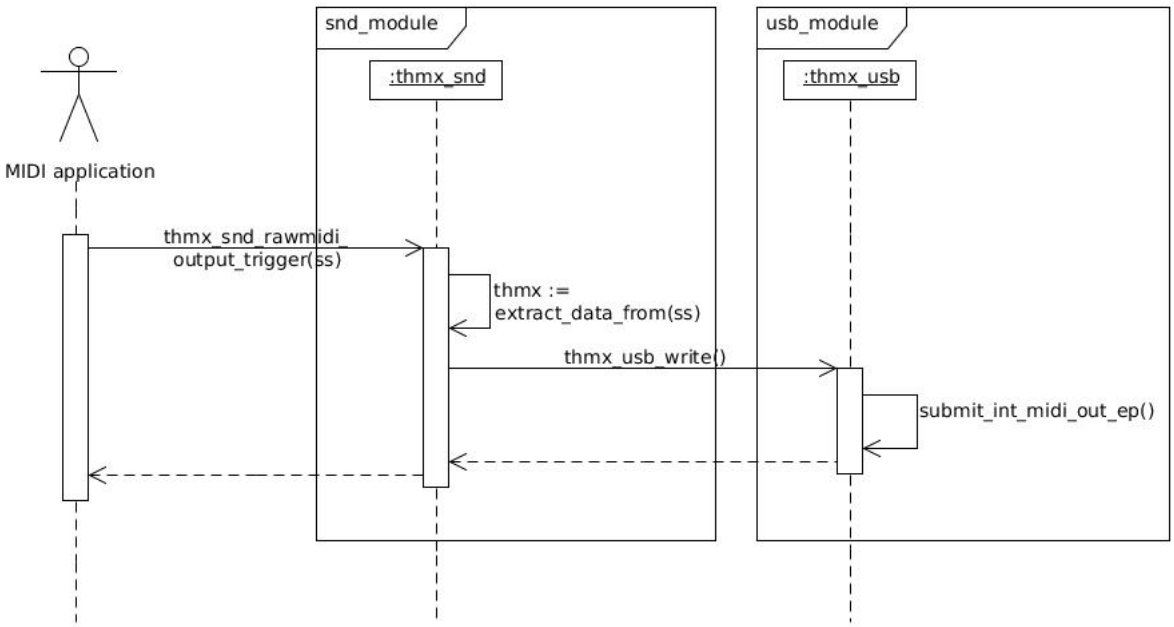

Figura 5.17: Diagramma della sequenza relativa all'invio di messaggi MIDI di output per dispositivi esterni

questa "write" viene implementato passandogli come argomento una `struct snd_rawmidi_substream`. All'interno di questa, come dati privati salvati al momento dell'inizializzazione, è stato inserito un puntatore alla struttura `thmx` di riferimento, tramite il quale è possibile ottenere la funzione di scrittura sul dispositivo usb(mostrata in Figura con il nome di `thmx_usb_write()`) implementata all'interno di `thmx_usb`. Tale funzione semplicemente inoltra i dati sull'endpoint specifico per il MIDI out.

Sequenze del sistema embedded

Passiamo ora a considerare il punto di vista del sistema embedded. Poiché l'API del kernel Linux relativa a questo sistema si riferisce ai "gadget", i vari prefissi dei nomi seguono questo stile, per cui, ad esempio, il modulo del kernel si chiama `gadget_module`.
In particolare, le funzioni di questo modulo vengono di solito iniziate con il prefisso `gthmx` dove, appunto, la 'g' sta per "gadget". La Figura 5.18 mostra la fase di inizializzazione del sistema da questo punto di vista. In maniera simile a quanto visto per il sistema embedded, anche qui l'oggetto centrale, `thmx_gadget`, implementa la funzione di inizializzazione principale (la `bind()`) e, dopo aver allocato le strutture dati, richiama le funzioni di inizializzazione degli altri moduli, impostando così gli endpoint, la porta seriale, il dispositivo a caratteri per l'I/O dal livello applicativo, gli elementi hardware programmabili e, infine, alloca anch'esso un kernel thread interno che permette di gestire i comandi di controllo che arrivano dal sistema host. Infatti, tali comandi arrivano dal callback che viene eseguito alla fine dell'URB su uno degli endpoint interrupt. Da questo contesto, generalmente è indicato non eseguire operazioni "arbitrarie" (e i comandi di controllo possono effettuare operazioni, in generale, non predicibili) per cui si è ritenuto meglio eseguirli in un thread separato.
Ma la fase di inizializzazione del sistema embedded viene in realtà completata quando il sistema host manda la richiesta di setup (Figura 5.19). Il sistema richiederà al driver i suoi descrittori, per poi confermare la configurazione inviata dal sistema embedded. Questo farà si che gli endpoint diventino effettivamente attivi, ma prima dell'inizio delle comunicazioni il sistema host richiederà i descrittori stringa del profilo di configurazione in modo da poterlo impostare lato host.
Una sequenza che, dal punto di vista del sistema embedded, diventa particolarmente semplice è l'invio del flusso audio sull'endpoint isochronous, mostrata in Figura 5.20. Una volta introdotta l'applicazione nello user space che legge i campioni audio appoggiandosi al driver ALSA standard, infatti, questa applicazione dovrà soltanto scrivere sul nodo del dispositivo a caratte-

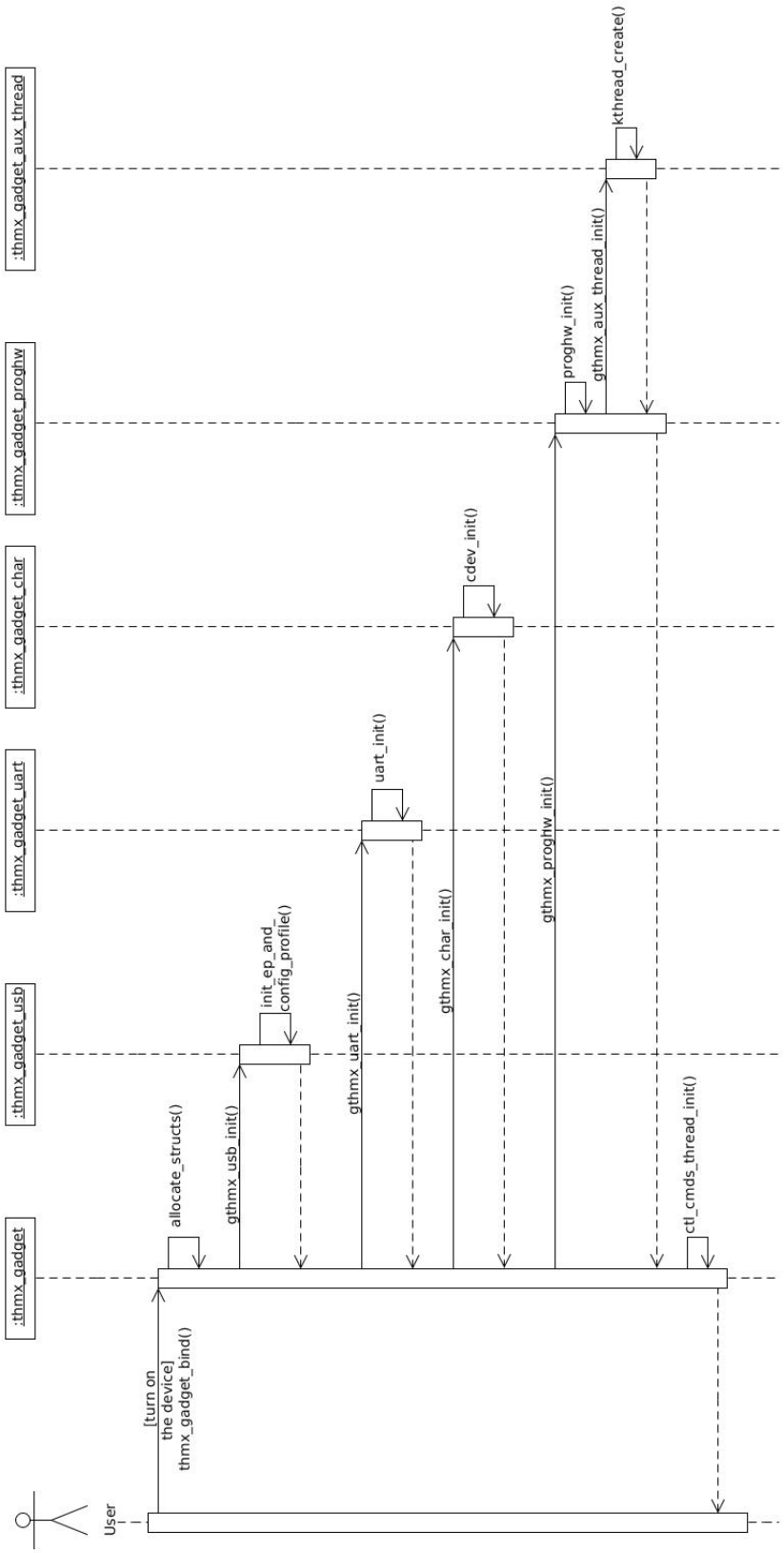

Figura 5.18: Diagramma della sequenza di inizializzazione del sistema embedded

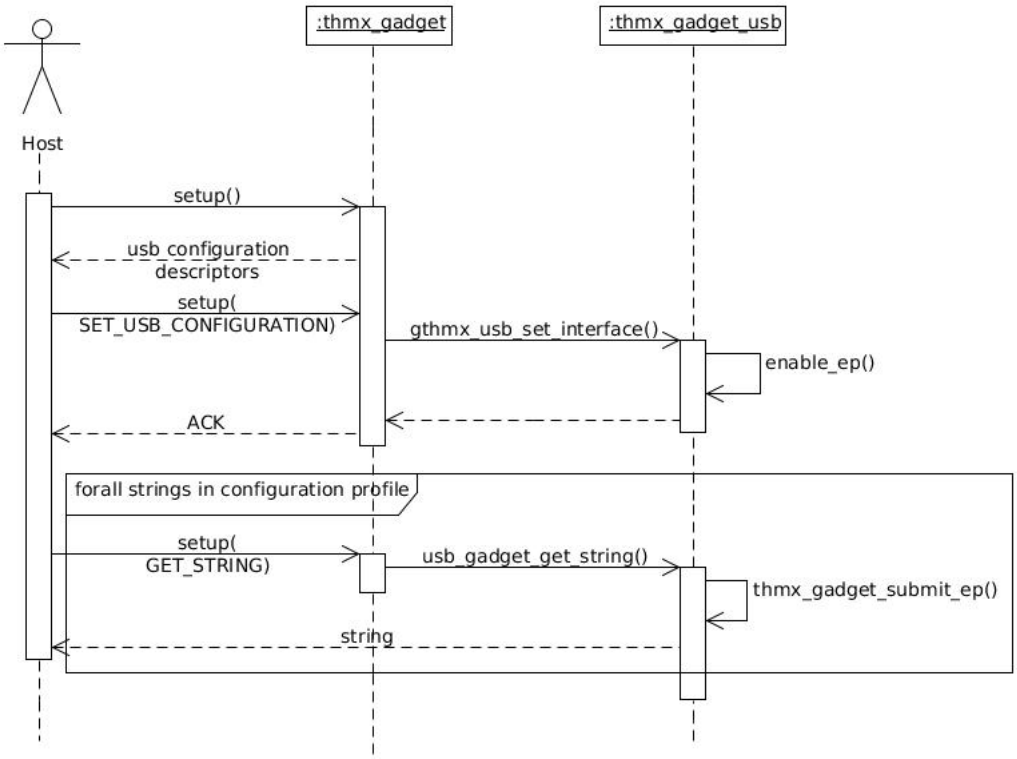

Figura 5.19: Diagramma della sequenza di setup del sistema embedded

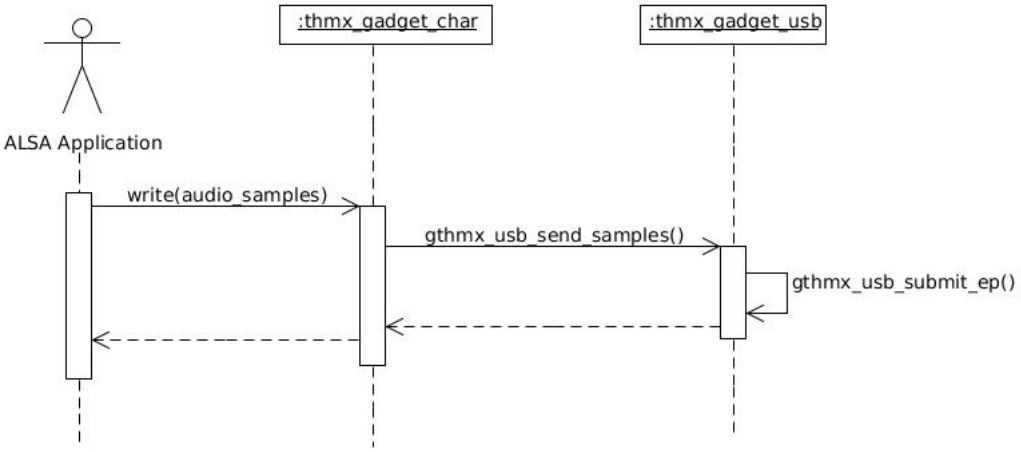

Figura 5.20: Diagramma della sequenza relavita all'invio di campioni audio al sistema host

ri creato dal driver, il quale provvederà a inoltrarlo sull'endpoint isochronous di input. In maniera simile, la Figura 5.21 mostra l'invio dei messaggi MIDI destinati ai dispositivi MIDI esterni. Come si nota, a parte la fase iniziale che viene implementata tramite un messaggio di `setup()` da parte dell'host, la sequenza è assai simile a quella in Figura 5.20. Naturalmente, qui la scrittura finale verrà effettuata non su un endpoint USB ma sulla porta seriale.
Infine, mostriamo le due sequenze relative all'esecuzione di comandi di controllo che vengano invocati da un interrupt hardware o dal software in esecuzione sull'host. La Figura 5.22 mostra il primo caso. L'oggetto `thmx_gadget_proghw`, responsabile dell'hardware programmabile, all'interno l'interrupt handler invocherà il kernel thread ausiliare incaricato dell'esecuzione del comando di controllo correntemente associato all'hardware.
Infine, la Figura 5.16 mostra la sequenza di operazioni per l'esecuzione di un comando di controllo invocato dall'host. Qui la richiesta arriva sull'endpoint interrupt adibito alla gestione di tali richieste, il quale semplicemente sveglia il thread interno all'oggetto `thmx_gadget` che si occupa dell'esecuzione dei comandi di controllo. Nel caso in cui il comando di controllo preveda l'invio di dati di ritorno, questo thread si occuperà di chiamare la funzione appropriata dell'oggetto `thmx_gadget_usb` per l'invio di tali dati.

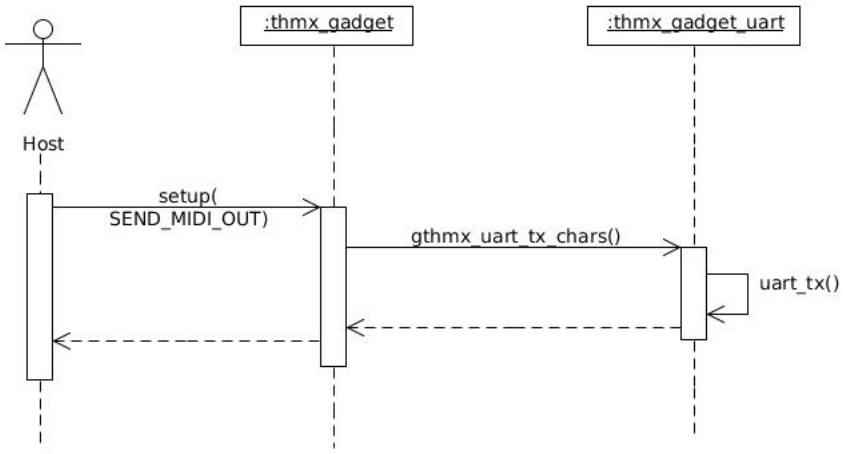

Figura 5.21: Diagramma della sequenza relavita all'invio di messaggi MIDI ad un dispositivo MIDI esterno

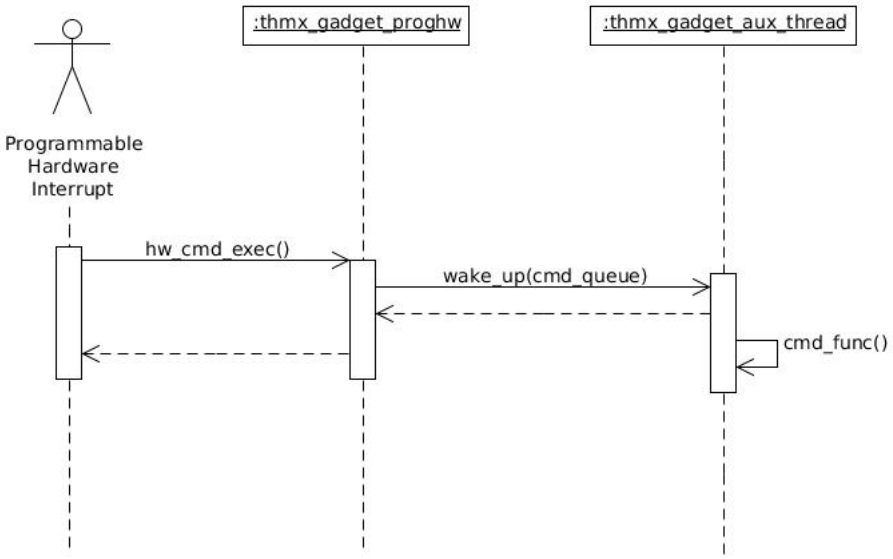

Figura 5.22: Diagramma della sequenza relavita all'esecuzione di un comando di controllo invocato da un interrupt hardware

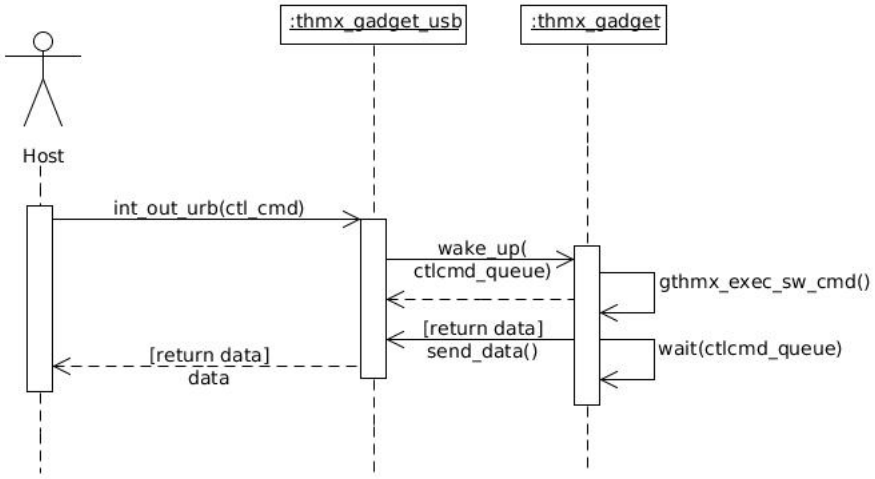

Figura 5.23: Diagramma della sequenza relavita all'esecuzione di un comando di controllo invocato dal lato host

Capitolo 6

Scelte di implementazione del sistema

Dopo aver trattato in maniera approfondita la progettazione "astratta" (per quanto non priva di piccoli test implementativi) del sistema, di seguito verranno forniti i dettagli di come il Thereminux è stato effettivamente implementato. Per fare questo, dapprima si presenterà la preparazione dell'ambiente dove il sistema è stato sviluppato ed eseguito (cfr. 6.1), spiegando quali strumenti sono stati utilizzati e come. Successivamente, si approfondirà il sistema vero e proprio (cfr. 6.2). Naturalmente, l'intero codice sorgente del sistema è stato allegato al materiale consegnato e documentato, oltre che all'interno degli stessi file sorgente, anche tramite un manuale di riferimento generato tramite *Doxygen*. Tale materiale è da considerarsi un valido ausilio alla trattazione che segue.

6.1 Ambiente di sviluppo ed esecuzione

L'ambiente hardware dove il sistema è stato sviluppato ed eseguito è composto da un notebook che ha funzionato ha host, la scheda Beagle Board come gadget e una scheda audio esterna, collegata tramite USB all'host, per la gestione dei flussi audio e MIDI. Dal punto di vista software, è stato utilizzata una distribuzione Linux Ubuntu 8.04 e il kernel 2.6.31 lato host, una versione base della distribuzione Angstrom [66] di Linux Embedded e il kernel 2.6.32 lato gadget.

6.1.1 Ambiente lato host

Per lo sviluppo del Thereminux, il sistema Linux configurato ad hoc (distribuzione Ubuntu 8.04) è stato ospitato su un computer host in una partizione dedicata. Il sistema host utilizzato è stato un pc con processore Intel Core Duo P8400. Di seguito viene data la descrizione dettagliata di come questo ambiente è stato impostato per il corretto funzionamento.

Configurazione audio

Il primo passo è stato quello di configurare la scheda audio principale (quella che si occupa della gestione di default dei dati audio/MIDI). È stata utilizzata per questo la scheda audio Audiophile USB [52].
Per la configurazione si è proceduto seguendo la documentazione su [53] e, all'interno del kernel, sul file */Documentation/sound/alsa/Audiophile-Usb.txt*. La scheda Audiophile, infatti, mette a disposizione diverse interfacce audio (analigiche e/o digitali), per cui occorre selezionare i giusti parametri da passare al modulo `snd_usb_audio`. La procedura seguita è la seguente:

1. Si spegne la scheda audio;

2. si smonta il modulo `snd_usb_audio` con il comando
   `modprobe -r snd_usb_audio`;

3. si collega nuovamente il modulo con i seguenti parametri:
   `modprobe snd_usb_audio index=1 device_setup=0x09`
   Il valore di `index` può variare a seconda del sistema. In quello di chi scrive, sebbene la Audiophile era la prima scheda audio del sistema, caricarla con `index=0` fa sì che essa non venga rilevata correttamente. Per cui si è scelto quello immediatamente successivo. Per quanto riguarda, invece, i dettagli sul parametro `device_setup` si veda il file detto all'interno della documentazione del kernel, dove si legge:
   device_ setup=0x09
   - 24bits 48kHz mode with Di disabled
   - Ai,Ao,Do can be used at the same time
   - hw:1,0 is not available in capture mode
   - hw:1,2 is not available
   Dove Ai, Ao, Di e Do significano rispettivamente Analog Input, Analog Output, Digital Input e Digital Output;

4. a questo punto, una volta caricata la scheda, si procede all'impostazione dei parametri di jack control, che derivano dalle caratteristiche della scheda come descritte dall'argomento `device_setup`:

   (a) Sample Rate: 48000
   (b) spuntare l'opzione "Force 16 bit"
   (c) spuntare l'opzione "Realtime" (usata tramite il kernel RT e l'impostazione di `set_rlimits`, come spiegato tra poco)
   (d) Input device: hw:1,1 (sul sistema di chi scrive)
   (e) Output device: hw:1,0 (sul sistema di chi scrive)
   (f) Altri parametri lasciati con i valori predefiniti

5. fatto questo, è stata utilizzata l'uscita per le cuffie della scheda come output audio;

6. in ultimo, il sistema audio è stato eseguito impostando i giusti parametri di priorità. Per fare questo, nel file */etc/set_ rlimits.conf* è stata aggiunta la linea
   `@daniele /usr/bin/qjackctl nice=-10 rtprio=90 memlock=1000000`
   (cfr. 3.7.2)
   e il programma qjackctl è stato eseguito utilizzando il comando `set_rlimits /usr/bin/qjackctl`.

Tutto quanto descritto è in realtà stato implementato in un unico script interattivo che permette di avviare tutte le operazioni descritte necessarie al setup del sistema audio, così da agevolarne l'utilizzo.

Compilazione kernel rt

È stato utilizzato il kernel 2.6.31.12, per il quale la patch rt è disponibile nel momento in cui si è sviluppato il progetto [56]. Per il resto la riuscita della compilazione del kernel ha richiesto:

1. disattivazione della configurazione relativa al modulo RT2500 che si trova su:
   `Device Drivers -> Network device support -> Wireless LAN -> Ralink driver support -> Ralink rt2500` (sia la versione PCI che USB): questo infatti causa un errore nella compilazione perché pare [54] sia incompatibile con il kernel rt;

2. Per il debug dello sviluppo, sono state impostate le seguenti configurazioni:

   (a) `Kernel Acking -> Debug preemtible kernel`;
   (b) `Kernel Acking -> RT Mutex debugging, deadlock detection`;

   (c) `Kernel Acking -> Lock Debugging: prove locking correctness`

   (d) `Kernel Acking -> Tracers -> Trace syscalls`

3. abilitazione del `Power Management Support -> ACPI Support` ma senza includere le seguenti sotto-configurazioni di questo:

   (a) `Button`

   (b) `Fan`

   (c) `Processor`

   (d) tutte le configurazioni nominate come "deprecated"

   Il fatto che ACPI vada comunque abilitato dipende dal fatto che si incontrano problemi nella compilazione e utilizzo dei hr-timer se questo è completamente disabilitato [55];

4. disabilitazione del `Power Management Support -> APM BIOS Support`.

Il file con la configurazione del kernel è stato incluso nel materiale consegnato, all'interno della directory *thereminux/thmx_src/utility_scripts/host/system_scripts*.

6.1.2 Ambiente lato gadget

Compilazione kernel e distribuzione Linux per la BeagleBoard

Per la compilazione del kernel lato gadget sono stati fatti alcuni tentativi di compilazione "a mano", risultando però in alcune instabilità del kernel e della distribuzione del sistema operativo quando questo veniva eseguito sulla Beagle Board. Infatti, si voleva evitare di utilizzare l'intera distribuzione Angstrom di Linux Embedded, in quanto mette a disposizione una quantità di strumenti che creano un completo ambiente interattivo da gestire tramite console mentre, per la nostra applicazione, poteva essere utile avere anche soltanto un sistema minimale che mettesse a disposizione una console con comandi base.
È stato utilizzato, allora, il sistema di sviluppo OpenEmbedded [63] e lo strumento BitBake [65], con il quale è stato possibile ottenenere un'interfaccia a più alto livello per la configurazione e la compilazione dei sorgenti del kernel e la costruzione di una distribuzione ad hoc per la nostra specifica architettura target. In particolare, questo sistema gestisce automaticamente tutte le varie dipendenze tra i pacchetti software da includere nella distribuzione costruita in maniera da garantire una corretta esecuzione del sistema sull'architettura target. Per questo, si è proceduto seguendo [63] per la configurazione iniziale

di OpenEmbedded sul sistema host. Una volta generato il kernel e la distribuzione, è stato implementato lo script `beagle-setup-sd` (cfr. 5.3.2) per la raccolta di queste informazioni e il loro inserimento nella scheda SD da usare con la Beagle Board.
In particolare, la configurazione in locale è stata effettuata seguendo i seguenti passi:

1. cancellazione della directory *~/.oe/** (se esiste);
2. dalla directory ${OE_BASE} (nel sistema locale è *~/beagleboard/oe/angstrom-build/angstrom-setup-scripts*) viene lanciato lo script `oebb.sh` messo a disposizione con OpenEmbedded (e modificato in modo da utilizzare le varie partizioni in locale). I comandi da lanciare per la configurazione sono:
   (a) (se si vuole partire con un sistema "pulito") cancellare tutto il contenuto delle directory *$OEBASE/sources*, *${OE_BASE}/build/conf/*, *${LOCAL_LARGE_DIR}/build/tmp* e *${LOCAL_LARGE_DIR}/sources/downloads*
   (b) `./oebb.sh config beagleboard`
   (c) `./oebb.sh update`
3. dalla $OEBASE lanciare il comando `. ~/.oe/environment`
4. usare `bitbake -c <comando>`
5. ogni volta che si vuole iniziare una sessione con bitbake, occorre richiamare, dalla $OEBASE, il comando `. ~/.oe/environment`. Questo imposta la directory corrente come base per le operazioni da eseguire con bitbake;
6. OpenEmbedded funziona specificando i vari pacchetti software da installare tramite dei file con l'estensione ".bb", di solito posizionati nella directory *$OE_SRC_BASE/recipes/* e nelle sue sotto-directory. partendo dalla directory di base delle configurazioni *$OEBASE* (nel sistema di chi scrive è *~/beagleboard/oe/angstrom-build/angstrom-setup-scripts/*), il file *build/config/local.conf* contiene le specifiche su dove scaricare i pacchetti, dove trovare i file .bb all'interno del file system, il nome della distribuzione di linux da ottenere e quello della macchina target. Impostando questi parametri correttamente il sistema otterrà dalla rete tutti gli opportuni pacchetti software;

7. fatto questo, si può iniziare la configurazione generale di tutto ciò che serve per ottenere i sorgenti del kernel da installare sulla macchina target tramite
   `bitbake -c configure virtual/kernel`
   dove `virtual/kernel` specifica il kernel migliore per l'architettura richiesta e specificata nel file di configurazione; l'interfaccia offerta da bitbake permette di eseguire una serie di comandi, come
   `bitbake -c clean <pacchetto>`
   `bitbake -c compile <pacchetto|target>`
   `bitbake -c deploy <pacchetto|target>`
   rispettivamente per il reset dell'ambiente, la compilazione e la "distribuzione" di un pacchetto (ad esempio, `linux-2.6.32`) o di un target generico. A questo proposito, su [64] vengono descritti diversi target standard forniti da OpenEmbedded per generare diverse tipologie di distribuzioni di base, ad esempio includendo o no le librerie grafiche, ecc. Come detto all'inizio del paragrafo, per il sistema embedded del Thereminux è stata utilizzata una distribuzione con le sole funzionalità di base;

8. successivamente è stata utilizzata una configurazione del kernel modificata rispetto a quella messa a disposizione dalla distribuzione Angtrom. Partendo da questa, sono state fatte le seguenti modifiche:

   (a) è stato disabilitato il supporto ai pulsanti GPIO
   `device_drivers->input_device_support->keyboards->gpio_buttons`
   in quanto caricava un modulo che registrava la linea di GPIO che corrisponde al pulsante programmabile della Beagle Board, non permettendo così al driver del Thereminux di prenderne possesso;

   (b) è stato abilitato il solo la modalità gadget nello stack OTG
   `device_drivers->usb_support->`
   `inventra_highspeed_dual_role_controller->`
   `driver_mode=usb_peripheral`
   infatti lo USB 2.0 permette di definire dei dispositivi USB OTG (On-The-Go), ovvero che riescono ad assolvere ad entrambe le funzioni sia di gadget (rispetto ad un host al quale sono connessi) sia di host (rispetto a dispositivi che a loro volta si connettono a loro). Sebbene la Beagle Board metta a disposizione la possibilità di questa modalità di funzionamento, non è stata utilizzata perché, per prima cosa, non serviva per la nostra applicazione e, secondo, si è visto che richiedeva di fornire una alimentazione supplemen-

tare alla scheda rispetto a quella che viene portata dal cavo USB, e questo non è sembrato desiderabile nel nostro caso;

(c) sono stati caricati come moduli tutti i gadget drivers `device_drivers->usb_gadget_support->usb_gadget_drivers=m` per il loro possibile utilizzo in fase di test.

Avvio BeagleBoard

Come già detto in precedenza (cfr. 5.3.2) è stato sviluppato uno script che permette di inizializzare una card SD/MMC formattandola con due diverse partizioni, una fat di avvio (dove andrà il boot loader U-Boot e l'immagine del kernel) e una ext3 (dove andrà il rootfs e i moduli). La formattazione può essere fatta in automatico utilizzando il tool in [61], che viene eseguito dallo script `beagle-setup-sd` implementato. Oltre a questo, lo script carica la distribuzione Angtrom customizzata che è stata sviluppata e predispone già il modulo `thmx_gadget` all'interno del sistema e crea una directory dove copia gli script di utilità che possono essere avviati, tramite seriale, sulla Beagle Board stessa a runtime. Infine, è stata seguita la procedura descritta in [62] per i dettagli sulla creazione della SD card e per l'inizializzazione della porta seriale tramite *minicom*.

Moduli e applicazioni compilati per la BeagleBoard

Per quanto riguarda la compilazione di applicazioni per la BeagleBoard, occorre semplicemente utilizzare un cross-compiler in modo da generare eseguibili per l'architettura ARM.
Invece, la compilazione di moduli del kernel richiede, naturalmente, per prima cosa i sorgenti del kernel di riferimento. Il kernel Angstrom [66] usa, di default, il kernel 2.6.32, per cui da lì si è partiti. Una volta ottenuti i sorgenti del kernel, si è lavorato con un Makefile adattato a questo. Ad esempio, immaginando di voler compilare un semplice modulo chiamato `hello` si può usare un Makefile come il seguente:

```
ifneq ($(KERNELRELEASE),)
        obj-m := hello.o
else
        KERNELDIR ?= <path_to_kernel_sources>
        PWD := $(shell pwd)
default:
        $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif
```

Successivamente, la compilazione tramite Makefile dovrà avvenire specificando l'architettura di riferimento e il *cross compiler* utilizzato, con il comando seguente (si assume che si stia utilizzando la toolchain CodeSourcery G++ Lite [60]):

```
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
```

Chiaramente, queste indicazioni sono state inserite in un'opportuno Makefile per la compilazione dei sorgenti del modulo.
Fatto questo, i moduli del kernel vanno inseriti nella directory */lib/modules/<kernel_ version>*. Affinché un modulo venga caricato all'avvio, è necessario inserire un file, contenente il nome del modulo, sotto la directory */etc/modutils*. Ad esempio, per caricare il modulo `hello` precedentemente compilato (ed inserito sotto */lib/modules/<kernel_ version>*) è stato sufficiente creare il file */etc/modutils/hello* che contiene, appunto, la stringa "hello", per poi richiamare il comando `update_modules`. Riavviando il sistema, il modulo viene caricato in fase di avvio.
Quanto descritto è stato incluso in un apposito script che viene posto nella home del sistema embedded e permette, al primo avvio dopo il setup della card SD, di effettuare tutte le impostazioni necessarie all'esecuzione del modulo `thmx_gadget`.

6.2 I device driver sviluppati

L'implementazione del Thereminux ha richiesto l'utilizzo di alcuni diversi sotto-sistemi e interfacce presenti nel kernel Linux nell'ambito dei device driver:

- lato host, è stato implementato un device driver USB e uno ALSA;
- lato gadget, in un unico modulo del kernel sono stati inclusi un device driver USB Gadget, uno a caratteri per la gestione dei campioni audio dal livello applicativo, uno UART per la gestione del MIDI-out e uno GPIO per l'acquisizione del segnale dal pulsante programmabile della Beagle Board.

Per la descrizione accurata di ognuna delle parti descritte, si è scelto di procedere idealmente come se si seguisse un "ciclo di vita" tipico del sistema: si partirà così dall'avvio (cfr. 6.2.1) fino ad arrivare alla disconnessione dello strumento (cfr. 6.2.9), passando da tutti i sotto-sistemi che implementano le varie funzionalità del sistema già presentate nel Capitolo 5.

6.2.1 L'avvio del sistema

Come descritto con le sequenze presentate (cfr. 5.4) la fase di avvio è piuttosto semplice ed è portata avanti dai due oggetti principali sia lato host che gadget (implementati, rispettivamente, da *host/usb_ module/thmx.c* e *gadget/thmx_ gadget.c*). Nel primo caso, la funzione `thmx_probe()` viene richiamata dal *core* USB per l'avvio del driver USB, e questa alloca il contenitore di dati principale del sistema (la `struct thmx`) che contiene i puntatori a tutte le altre strutture dati relative agli altri oggetti necessari all'esecuzione. Tra i campi di questa struttura, viene allocata la coda di attesa `thmx.wait_queue` che viene utilizzata nel sistema per implementare degli sleep basati su timeout (tramite la funzione `wait_event_interruptible_timeout()`). Questi sleep sono utilizzati per effettuare una sincronizzazione sia quando l'host deve ricevere dati dal gadget (come si vede nella funzione `thmx_usb_send_ra_req()` in *host/usb_ module/thmx_ usb.c*: l'attesa qui permette di essere certi che questi dati sono stati inviati) sia di inviare messaggi MIDI al livello applicativo (come in `call_vtm_conversion()` dentro *host/usb_ module/thmx_ aux_ thread.c*: l'attesa permette di far sì che ALSA interpreti i messaggi come avvenuti in momenti differenti, mentre inviare un unico buffer fa sì che i messaggi sono interpretati come avvenuti nello stesso istante). Per il resto, ogni oggetto viene inizializzato con una sua funzione specifica, allocando tutte le risorse che gli sono necessarie come verrà spiegato in seguito.
Sul gadget, la situazione è molto simile: il sotto-sistema USB, in questo caso, richiama la funzione `gthmx_bind()` dichiarata nel driver usb (*gadget/thmx_ gadget.c*), la quale si occupa di inizializzare tutti gli oggetti contenuti nella `struct thmx_gadget` richiamando le loro funzioni specifiche.

6.2.2 La comunicazione USB

La comunicazione tra host e gadget tramite USB è stata implementata seguendo quanto già descritto in precedenza (cfr. 3.3.5 e 3.7.1). Per prima cosa il gadget dichiara staticamente (nel file *gadget/thmx_ gadget_ usb.c*) tutti i descrittori necessari a rispondere correttamente alle richieste dell'host. Tra questi (cfr. 3.2) il device descriptor, il config descriptor, l'interface descriptor, i vari descrittori per gli endpoint e per le stringhe del file di configurazione. Inoltre, per quanto riguarda gli endpoint, come descritto nella Sezione 3.2 lo standard USB 2.0 richiede che, quando un dispositivo vuole presentarsi con una configurazione High-Speed (cfr. 3.2), e questo è il caso del Thereminux, deve comunque dichiarare anche i descrittori adatti per la comunicazione Full-Speed, per cui nel file *gadget/thmx_ gadget_ usb.c* troviamo le varie coppie di

descrittori per ogni endpoint. Inoltre, troviamo la dichiarazione del device qualifier che espone le caratteristiche della configurazione in High-Speed che il Thereminux utilizza come preferenza.

Inizializzazione

In fase di inizializzazione delle strutture di comunicazione USB (vedi la funzione `gthmx_usb_init()` in *gadget/thmx_ gadget_ usb.c*), il gadget verifica se il dispositivo sul quale è in esecuzione è compatibile con la modalità High-Speed tramite la funzione `gadget_is_dualspeed()` e, se questo è vero (come nel nostro caso) inizializza i descrittori High-Speed dichiarati per gli endpoint. Per quanto riguarda la comunicazione, è l'oggetto principale del modulo (vedi *gadget/thmx_ gadget.c*) ad occuparsi di acquisire le richieste standard e a chiamare le apposite funzioni per provvedere alle risposte. In questo modo, l'oggetto `thmx_gadget` riceve le richieste di setup standard (funzione `stardard_setup_req()`) e ne effettua il parsing, valutando il valore dei vari campi della richiesta (cfr. 3.2.3), come il `bRequest` per il nome della specifica richiesta e il `w_value` contenente l'argomento opzionale passato insieme alla richiesta. Tra le funzioni più importanti richiamate all'interno di questo parsing, c'è `gthmx_usb_req_send_desc()` (definita in *gadget/thmx_ gadget_ usb.c*), che tra i suoi parametri prende in argomento il tipo di descrittore da consegnare e lo ritorna. Questa funzione è anche usata per inviare i descrittori stringa relativi al profilo di configurazione.
Dal punto di vista dell'host, la situazione è più semplice. Infatti, il sottosistema USB del kernel si occupa di inviare queste richieste di setup al gadget e di costruire la `struct usb_interface` che viene passata alla funzione `thmx_probe()` descritta sopra. Nell'inizializzazione dell'oggetto usb (vedi la funzione `thmx_usb_init()` in *host/usb_ module/thmx_ usb.c*) tale struttura dell'interfaccia USB viene usata per verificare che il dispositivo connesso, seppure dichiara di poter essere gestito da questo driver tramite il suo vendorID e productID, effettivamente mette a disposizione tutti gli endpoint necessari per l'esecuzione. In tal caso, vengono allocate tutte le strutture dati per la gestione di questi endpoint, nonché viene inizializzato il campo `usb_ops->write` interno alla struttura `thmx_usb`: come descritto in precedenza (cfr. 5.3.6 e 5.4), infatti, il modulo `thmx_snd`, all'arrivo di dati di MIDI-out sul suo nodo ALSA, usa questa funzione per inoltrare tali messaggi all'oggetto USB del Thereminux, che si occuperà di inviarli al gadget.
Infine, l'inizializzazione della comunicazione USB termina con l'inizializzazione e l'avvio del kernel thread che si occupa della ricezione dei campioni audio dall'endpoint isochronous. Le funzioni relative a queste operazioni sono contenute nel file *host/usb_ module/thmx_ usb.c*.

Comunicazioni sugli endpoint

Il thread di ricezione semplicemente definisce una routine principale (`thmx_usb_receive()`) che gestisce la lettura sull'endpoint isochronous. Come vedremo più avanti (cfr. 6.2.6) tale endpoint rappresenta una risorsa condivisa, in quanto viene utilizzato anche per la ricezione della stringa di debug che viene mandata dal gadget in seguito ad un opportuno comando di controllo. Per prevenire conflitti in queste comunicazioni, la prima cosa che il thread di ricezione fa è acquisire un lock relativo proprio all'endpoint isochronous, lock che viene mantenuto per l'intera durata delle ricezioni di pacchetti audio. Naturalmente, il processo che tenta di eseguire il comando di controllo relativo allo stesso endpoint non resterà bloccato in attesa che il lock si liberi (teoricamente il thread di ricezione potrebbe mantenerlo per l'intera esecuzione del sistema), ma utilizzerà la funzione `down_trylock()` per verificare se la risorsa è libera. La funzione `thmx_usb_iso_read()` esegue la lettura vera e propria, allocando un opportuno URB e inviandolo al dispositivo. Per l'invio, l'endpoint isochronous permette di specificare che vengano inviate più richieste contemporaneamente al dispositivo, così da garantire che il flusso di dati possa essere acquisito il prima possibile. Tale meccanismo viene specificato anche nella ricezione fatta dall'oggetto `thmx_usb`, con l'inizializzazione seguente:

```
urb->dev = usb->udev;
urb->context = dev;
urb->pipe = usb_rcvisocpipe(usb->udev,
                            iso_in->iso_in_endpointAddr);
urb->interval = 16;
urb->transfer_flags = URB_ISO_ASAP;
urb->transfer_buffer = iso_in->iso_in_buffer;
if (!urb->transfer_buffer) {
   retval = -EINVAL;
   goto error;
}
urb->complete = thmx_read_iso_callback;
urb->number_of_packets = NUM_ISO_PACKETS;
urb->transfer_buffer_length = iso_in->iso_in_size;
for (i=0; i < NUM_ISO_PACKETS; i++) {
   urb->iso_frame_desc[i].offset = i * iso_in->maxpacket;
   urb->iso_frame_desc[i].length = iso_in->maxpacket;
}
```

Come si nota, viene impostata la struttura `urb->iso_frame_desc` che descrive i vari frame che verranno acquisiti, uno per pacchetto, e verranno ricostruiti quando l'URB ritorna. Tale terminazione della richiesta è gestita dalla funzione di callback, `thmx_read_iso_callback()`. Questa effettua delle importanti operazioni: controlla il numero di pacchetti ricevuti e, per ognuno, copia il contenuto dentro il buffer circolare che verrà poi usato per la conversione Voltage-To-Midi (cfr. 6.2.5). Inoltre, questa funzione effettua alcuni controlli importanti per la comunicazione: in caso di errori gravi nell'URB oppure se non vengono ricevuti pacchetti continuativamente per un tempo troppo lungo, l'esecuzione verrà passata nuovamente alla routine principale del thread, la quale invierà al gadget una richiesta per cercare di effettuare un recovery in caso di errore, si bloccherà in un'attesa di alcuni secondi e riproverà a ricevere dati.
Da parte sua, il gadget riceverà questa richiesta come un comando di controllo (cfr. 6.2.6) e capirà che l'host non sta ricevendo dati audio. Ora, se è il gadget stesso che non sta più mandando dati (perché l'applicazione ALSA nello user-space per la cattura dei campioni audio è stata bloccata) il controllo restituirà all'host la stringa `ISOCH_IN_STOPPED` e non effettuerà altre operazioni. Invece, nel caso lui si accorga che è attualmente in corso l'invio di dati, qualcosa non va, perché l'host dovrebbe riceverli e, invece, il suo messaggio indica che questa ricezione non avviene. In questo caso, il recovery implementato semplicemente prova a disabilitare (tramite la funzione `usb_ep_disable()`) e a riabilitare (usando `usb_ep_enable()`) l'endpoint, inviando poi all'host la stringa `ISOCH_IN_RESTART`.
Oltre a questo, per garantire la comunicazione su tutti gli endpoint, il gadget, in fase di inizializzazione, avvia una richiesta pendente sugli endpoint di interrupt usati per l'output (ovvero per le comunicazioni dall'host al gadget), così che quando arrivano dati questi effettivamente vengano ricevuti. Invece, per le comunicazioni di input sarà l'host ad inviare richieste, mentre il gadget eseguirà l'invio soltanto quando effettivamente ci sono dati da mandare. Infine, per quanto riguarda la possibilità di conflitti sui dati inviati sull'enpoint isochronous, il gadget non si occupa di gestire la mutua esclusione della risorsa come l'host, ma si limita a rispondere a richieste che vengono già fatte dall'host rispettando tale sincronizzazione.

6.2.3 La gestione di ALSA lato host

Sull'host, l'interfaccia ALSA del Thereminux è gestita interamente dal modulo `thmx_snd` (vedi il file *host/snd_ module/thmx_ snd.c*).

Inizializzazione

Visto che questo modulo fornisce delle funzionalità liberamente accessibili da altri moduli che sono costruiti sull'interfaccia del Thereminux (ovvero, che usano come struttura principale la `struct thmx`) è stato necessario adoperare un po' di attenzione in fase di inizializzazione. Infatti, ogni dispositivo basato su thmx che cerca di registrarsi ad ALSA viene associato ad una sound card che viene allocata nel sistema e, per questa, c'è bisogno di un identificativo univoco. Esso è dichiarato come variabile intera statica e globale all'interno del modulo, così che venga incrementata ogni volta che un modulo di basso livello (come lo `usb_module` del Thereminux) richiama la funzione `thmx_snd_probe()`. Per questo, assieme all'indice increntale, viene mantenuto anche un mutex statico (`snd_sem`) che viene acquisito per il tempo necessario all'inizializzazione del dispositivo audio da parte del modulo. Tale inizializzazione viene effettuata dalla funzione `thmx_create_snd_device()`. Essa integra quanto già presentato per il modulo di test *minisnd* (cfr. 3.5), allocando una sound card e associando ad essa i dispositivi MIDI richiesti. Per questo punto, è il driver di basso livello che, in base al valore letto dal profilo di configurazione inviato dal dispositivo, imposta questa variabile (di default il Thereminux mette a disposizione il MIDI-out attivo).

Comunicare con le applicazioni ALSA

La comunicazione tra il dispositivo e il livello applicativo, a questo punto, è interamente gestita dalle funzioni relative agli stream MIDI. Come si legge in [6], per ogni dispositivo MIDI è necessario implementare, oltre alle normali open e close, tre funzioni che vengono richiamate dal *core* ALSA in fase di inizializzazione del dispositivo:

1. la `rawmidi_input_trigger` (implementata con la `thmx_snd_rawmidi_input_trigger()`) viene eseguita quando viene attivata/disattivata l'interfaccia di ricezione MIDI dal dispositivo. In pratica, il *core* ALSA richiama questa funzione passando il substream di MIDI-input allocato sul dispositivo e un flag che indica se si tratta di una attivazione o disattivazione. Il driver, a questo punto, mantiene lo stato corrente tramite un bit in una apposita bitflag (`thmx_snd.rxtx_enable_flag`) e può valutare se è possibile inviare dati al livello applicativo (questo verrà comunque fatto quando ci sono messaggi MIDI pronti, ovvero dopo la conversione Voltage-To-MIDI);

2. la `rawmidi_output_trigger` (implementata con la `thmx_snd_rawmidi_output_trigger()`) fa la stessa cosa descritta al

punto precedente per quanto riguarda l'output. In particolare, però, essendo questa funzione relativa a dati che arrivano dal livello applicativo verso il driver, in caso di attivazione dell'output la funzione già esegue la reale ricezione dei dati. Per fare questo, richiama la funzione ALSA `snd_rawmidi_transmit_peek()` con cui acquisisce i messaggi MIDI e, successivamente, chiama la `thmx_snd_rawmidi_send()`, che semplicemente utilizza la funzione di `write()` dichiarata dall'oggetto usb del Thereminux (cfr. 6.2.2) per inoltrare dati sulla usb;

3. la `rawmidi_output_drain` (implementata con la `thmx_snd_rawmidi_output_drain()`) si occupa si svuotare la coda dei messaggi MIDI di output ancora da inviare al dispositivo prima della reale disconnessione presso ALSA. In realtà, nel caso del Thereminux si è visto che questa funzione serve a ben poco, visto che, in fase di disconnessione, il gadget è sparito e quindi non può ricevere altri dati.

Creazione ed eliminazione dinamica del MIDI-out

Una volta che il dispositivo MIDI è stato creato presso ALSA, l'API ALSA non permette di inserire o eliminare dinamicamente le sue parti che, in fase di esecuzione, vengono registrate e mantenute fino alla disconnessione. Per questo, l'implementazione della gestione dinamica del MIDI-out ha dovuto eseguire prima una disconnessione dell'intera sound card ALSA, per poi ricreare un nuovo dispositivo (che però mantiene l'indice incrementale di quello vecchio). Per la creazione, viene usata la stessa funzione (`thmx_create_snd_device()`) che viene richiamata in fase di inizializzazione. Anche qui, però, occorre prestare attenzione alle risorse condivise: infatti, il `thmx_aux_thread`, che viene risvegliato dal thread di ricezione di dati audio interno all'oggetto `thmx_usb`, richiama la conversione Voltage-To-MIDI dei dati ricevuti e poi chiama la `thmx_snd_rawmidi_receive()` che inoltra questi dati al livello applicativo usando la funzione standard di ALSA `snd_rawmidi_receive()`. Dunque, si collega al dispositivo MIDI di input per le sue operazioni e, prima di farlo, acquisisce un lock specifico di questa risorsa e che, appunto, deve essere preso anche in fase di riconfigurazione del MIDI-out. In questo modo, quando tale riconfigurazione sta avvenendo, si è sicuri che il thread che inoltra i messaggi MIDI al livello applicativo non può accedere al dispositivo di MIDI-input. In particolare, non viene consentito a tale thread di rimanere bloccato in attesa che la risorsa si liberi: questo perché, in fase di acquisizione continua di dati audio, rimanere bloccati su un buffer inevitabilmente porterebbe ad inviare dati obsoleti rispetto a quelli attualmente ricevuti. In questo senso, si è preferito optare per la possibilità di

perdere qualche pacchetto di dati ma mantenere una risposta il più possibile vicina a ciò che l'utente sta suonando in tempo reale.

6.2.4 La gestione di ALSA lato gadget e l'interfaccia a caratteri

Dal lato del gadget, come già detto (cfr. 5.3.6), si è preferito non implementare nuovamente un driver, già esistente, che semplicemente riceva dati dal convertitore audio della Beagle Board, ma si è invece utilizzato il driver ALSA già esistente e si è modificata l'utility *aplay*, disponibile insieme alle *alsa-utils* [32], per far sì che essa, una volta catturati i dati audio, li inoltri sul nodo */dev/thmx_gadget* presente nel file system. Il programma `thmx_gadget_alsa_capture`, presente nella directory *thmx_src/utility_scripts/gadget*, è una versione leggermente modificata di *aplay* in cui, rispetto a quest'ultimo, sono stati eliminati alcuni controlli nell'acquisizione audio, come il limite imposto ai file registrati che è stato eliminato consentendo di inviare dati indefinitamente, o la pulizia del file su cui scrivere i dati prima dell'inizio della registrazione (fatta da aplay per salvare ogni volta un nuovo file audio) che è stata saltata perché, altrimenti, lo script avrebbe eliminato il nodo del dispositivo all'interno del file system. Inoltre, la `thmx_gadget_alsa_capture` non consente di scegliere tra l'acquisizione di dati audio e la loro riproduzione (come invece *aplay* fa) ma mette a disposizione solo la prima funzione. Assieme a questo programma, lo script `thmx_alsa_capture_start` è stato inserito tra gli script di avvio del sistema e fa sì, dunque, che non appena terminato l'avvio del sistema operativo, invece di avere il prompt per il login, il sistema cominci semplicemente a ricevere dati audio inoltrandoli sul driver. Infine, comunque, l'esecuzione di questa registrazione può essere interrotta dall'utente (tramite `ctrl + C`) e il sistema consentirà il normale login. In questo caso, il sistema host smetterà di ricevere dati e inizierà ad inoltrare richieste di recupero (come spiegato nel paragrafo 6.2.2), per cui è stato messo a disposizione, nel file system del gadget, un comando (essenzialmente identico a quello per far partire la registrazione in fase di avvio del sistema) che permette di ricominciare l'acquisizione di dati audio e il loro invio all'host.

Il dispositivo a caratteri

Il nodo sotto */dev/thmx_gadget* viene collegato ad un driver a caratteri per un dispositivo virtuale (implementato dall'oggetto `thmx_gadget_char` nel file *gadget/thmx_gadget_char.c*), che sostanzialmente si occupa di prendere i dati audio attraverso la system call `write()` invocata dallo script

nello user-space e inoltrarli all'oggetto `thmx_gadget_usb`. Per la comunicazione tra questi due oggetti, è stata utilizzata una singola `kfifo` definita in `<linux/kfifo.h>`, ovvero un buffer circolare che, nello specifico della nostra applicazione, è stato utilizzato da un singolo scrittore (l'oggetto `thmx_gadget_char`) e un singolo lettore (l'oggetto `thmx_gadget_usb`). Usando questo meccanismo è stato possibile implementare il protocollo senza il bisogno di utilizzare lock tra i diversi oggetti (come si legge nella documentazione nel caso di un solo scrittore e un solo lettore), semplificando notevolmente il codice.
La funzione `thmx_char_write()` (che implementa la system call `write()`), inoltre, si occupa di gestire eventuale "modifiche" (causate dall'esecuzione di comandi di controllo) da apportare ai campioni audio. Come spiegato nel paragrafo 5.3.4, tali modifiche sono la possibilità di azzerare i campioni audio di input o quella di moltiplicarli per una costante. Attualmente, tali modifiche vengono applicate soltanto per il tempo di una singola esecuzione della `thmx_char_write()`.

Invii ritardati

Infine, è possibile immaginare che, nel caso in cui l'applicazione `thmx_gadget_alsa_capture` venga terminata dall'utente, il buffer di campioni acquisito dal driver può essere stato riempito soltanto in parte. Per come è implementata la funzione `thmx_char_write()`, la comunicazione di dati da inviare all'oggetto `thmx_gadget_usb` viene effettuata soltanto quando il buffer è pieno, per efficienza. In questo caso, la parte del buffer non inviata andrebbe probabilmente persa.
Per evitarlo, è stata utilizzata una `workqueue`, che ha permesso di creare uno speciale thread la cui routine (un `delayed_work`) è associata ad un timeout: se il timeout scatta, significa che non sono stati inviati dati per troppo tempo e, dunque, è il momento di inoltrare i dati rimasti all'interno del buffer (se ce ne sono) all'oggetto `thmx_gadget_usb`. Il timer della workqueue viene bloccato ogni volta che si entra nella funzione `thmx_char_write()` (così da evitare invii contemporanei) e riattivato al suo termine. Purtroppo, anche qui la documentazione fornita dai testi [5] [4] consultati, riferendosi al kernel 2.6.11, è risultata obsoleta rispetto al kernel utilizzato. Per questo, è stato consultato l'articolo [44] ed il driver *drivers/usb/misc/ftdi-elan.c*, da cui si è tratto un esempio di utilizzo dell'API.

6.2.5 La conversione Voltage-To-MIDI

Il modulo `vmt_module` (implementato all'interno del file *host/vtm_ module/thmx_ vtm.c*) mette pubblicamente a disposizione del kernel una singola funzione di utilità (oltre a quella di inizializzazione e disconnessione) che, dato in input un puntatore ad una `struct thmx`, estrae da tale struttura un buffer di campioni audio ed effettua la conversione Voltage-To-MIDI su di esso. Per l'acquisizione del buffer è stata utilizzata, come nel caso descritto nel paragrafo 6.2.4, una `kfifo` in cui l'unico scrittore è il `thmx_usb` e l'unico lettore è `thmx_vtm`. Come già detto (cfr. 3.7.3), tale operazione ha comportato l'utilizzo della FPU hardware presente sul sistema host e, dunque, in fase di inizializzazione viene effettuato un controllo sulle caratteristiche della CPU del sistema sul quale si sta eseguendo il driver: se la FPU non è presente, il caricamento del modulo `thmx_vtm` non sarà possibile, cosa che farà ritornare l'intera procedura di inizializzazione del sitema con errore. Nel caso in cui la piattaforma hardware sia compatibile con l'esecuzione, è possibile avviare la procedura di conversione VTM. Per fare questo, è stato necessario approfondire lo studio della libreria *aubio* già presentata (cfr. 3.7.3 e [48]), portando alla necessità di effettuare alcune importanti operazioni sul flusso di dati audio in input: infatti, le normali applicazioni nello user space, tra cui quelle che usano la libreria *aubio*, lavorano con file in formati complessi (come il WAV) e non con i byte grezzi. Perché un file audio RAW possa essere interpretato, infatti, è necessario specificarne il formato di codifica audio: i campioni audio possono essere codificati in molteplici modi, ovvero usando rappresentazioni a 8, 16 o 24 bit con o senza segno e disposte in Little Endian o Big Endian. I motivi dei tanti formati, chiaramente, è essenzialmente l'esistenza sul mercato di molteplici standard commerciali che si sono diffusi nel tempo. Poiché, come si legge in [48], il sistema Pitch-to-MIDI messo a disposizione dalla libreria *aubio* lavora con file in formato WAV, è stato necessario adoperare lo stesso formato ovvero una rappresentazione di campioni a 16 bit con segno, Little Endian (codificata in *aplay* con la stringa `S16_LE`).
Una volta arrivati questo buffer di byte grezzi, quindi, per prima cosa è stato effettuato un parsing di questi dati, da cui si ottiene un array di parole di 16 bit con segno (per i quali normalmente si utilizzano degli `short`). Oltre a questo, è necessario ottenere i valori floating point che sono rappresentati da queste parole.
Studiando il codice della libreria *aubio* ci si è accorti che essa si poggia su un'altra libreria, utilizzata pressoché ovunque nelle applicazioni audio per Linux: la *libsndfile* [67]. In particolare, è stato notato che la lettura di valori float da uno stream di byte interi viene fatta normalizzando i

valori tra -1 e 1 usando una costante specifica. Ci si è basati allora sugli esempi presenti per mostrare il funzionamento di aubio (vedi il file *aubio_ dir/examples/aubiopitch.c*) e si è arrivati a capire che la lettura viene fatta tramite una funzione `read_float()` che, nel caso di uno stream wav, viene inizializzata nel file *pcm.c* (dentro i sorgenti di libsndfile, vedi *libsndfile_ dir/src/pcm.c*) e, in particolare, dentro la funzione `pcm_init()`. Qui, a seconda del formato dei dati, viene usata una funzione `read_float()` differente. Nel nostro caso, poiché riceviamo dati a 16 bit (`short`) little-endian, viene usata la funzione `pcm_read_les2f()`. Questa, preso lo stream di signed-int, lo trasforma in array di float usando la normalizzazione detta. Lo studio effettuato ha permesso di adattare tutto questo al nostro caso e modificare la funzione di decodifica dello stream di byte dentro l'oggetto `thmx_vtm` (con la funzione `raw_bytes_to_smpl()`). Fatto questo, è stato ottenuto un buffer di dati floating point che è stato passato alla funzione `thmx_aubio_pitchyin_do()` per il riconoscimento del pitch del segnale e, successivamente, alla `thmx_aubio_freqtomidi` per l'estrazione della nota MIDI relativa al pitch ottenuto.

Generazione di messaggi MIDI

Una volta ottenuta una nota MIDI derivante dalla conversione Voltage-To-MIDI, è stato comunque necessario leggere il valore del *playingMode* dentro il profilo di configurazione del thmx. In base al valore ottenuto, sono stati generati due diversi stream di messaggi MIDI:

1. se il *playingMode* è impostato sulla stringa "staccato", verranno generati soltanto messaggi di Note On (cfr. 3.4.1), uno per ogni nota riconosciuta dalla conversione;

2. invece, se nel *playingMode* è specificato "glissando", per ogni nota viene generato uno stream di quattro messaggi MIDI che cercano di dare un effetto di transizione "morbida" da una nota all'altra:

   (a) per prima cosa, viene risuonata l'ultima nota suonata (messaggio di Note On);

   (b) a questa, è applicato un *Pitch Bend*, ovvero un glissando sul suono della nota, così che si avvicini alla nota successiva da suonare;

   (c) arrivati al massimo effetto di bending per collegare le due note, viene suonata la nuova nota e rilasciato il pitch bend. Questo comporta l'invio di due messaggi contemporaneamente: uno di Note On con la nuova nota e uno di Pitch Bend rilasciato. Il

Figura 6.1: Un esempio di *Pitch Wheel* installata su un noto sintetizzatore audio

senso di quest'ultimo messaggio sta nell'utilizzo classico con cui è nato: nella tastiere elettroniche, infatti, è quasi sempre presente una manopola a forma di ruota (Figura 6.1). Questa ha uno stato di riposo e può essere spostata sia verso l'alto (producendo un innalzamento del pitch della nota) sia verso il basso (con una conseguente diminuzione del pitch). Il messaggio MIDI di Pitch Bend (anche detto, a causa di questa immagine, di *Pitch Wheel*, cfr. 3.4.1) è stato creato proprio per l'utilizzo di questo dispositivo hardware.

I messaggi MIDI così prodotti, sono poi scritti dentro un apposito buffer che viene letto dal modulo `thmx_aux_thread` (che aveva richiamato la funzione di conversione Voltage-To-MIDI) e inviato al livello applicativo tramite le funzioni ALSA messe a disposizione di `thmx_snd`. In realtà, per ottenere il giusto effetto descritto poco fa, per l'invio dei messaggi di Pitch Bend il `thmx_aux_thread` implementa una pausa tra il primo messaggio di Note On e il primo Pitch Bend (così che vengano interpretati come messaggi arrivati in momenti differenti), mentre invia in un unico buffer i successivi due messaggi (così che ALSA li interpreti come avvenuti nello stesso istante). Tale pausa è implementata tramite una chiamata a `wait_event_interruptible_timeout()`.

6.2.6 I comandi di controllo dell'esecuzione

All'interno dell'oggetto `thmx_usb` è stata creata la funzione `thmx_ioctl()` per la gestione dei comandi di controllo dell'esecuzione, gestiti come comandi ioctl inviati dallo user-space. Per implementare dei comandi ioctl, bisogna

considerare che essi devono essere identificati come valori numerici univoci all'interno dell'intero sistema, per evitare che venga inviato un certo comando al dispositivo sbagliato e che questo, riuscendolo ad interpretare correttamente, esegua un'operazione magari dannosa [5]. Per prima cosa, dunque, è stato necessario allocare un *magic number* (univoco all'interno del sistema) che rappresenta un primo bitfield a cui vengono aggiunti dei valori numerici progressivi, così da costruire i comandi ioctl finali per il dispositivo. Consultando, all'interno del kernel, il file *Documentation/ioctl/ioctl-number.txt*, si è visto che `'x'` è, al momento della scrittura di questo documento, un magic number libero. Successivamente, sono state create le ioctl per i vari comandi. La funzione `thmx_ioctl` esegue il parsing degli argomenti passati in input e, se è stato inviato un comando valido, richiama una delle due funzioni `thmx_usb_send_ctl_cmd()` o `thmx_usb_send_ra_req()` per eseguire realmente il comando. Di queste due funzioni, la prima si occupa di prendere il comando e i suoi eventuali argomenti, costruire un URB sull'endpoint di output di tipo interrupt generico e inviare il comando. La seconda funzione, invece, implementa quelle che sono state chiamate richieste *Request/Answer* (RA Requst), ovvero che prevedono l'invio di un comando al gadget e una risposta da parte di quest'ultimo. A parte il comando di controllo per l'invio da parte del gadget di una stringa di debug sull'endpoint isochronous, tutte le altre richieste RA ricevono la risposta dal gadget sull'endpoint di input di tipo Bulk. Di seguito forniamo la lista dei comandi eseguiti da ognuna delle due funzioni descritte:

- comandi per `thmx_usb_send_ctl_cmd()`:
  - `ISO_IN_ZERO`: azzeramento dei campioni audio di input;
  - `ISO_IN_MULT`: moltiplicazione dei campioni audio di input per una costante;
  - `SET_MIDI`: impostazione del MIDI-out dello strumento;
  - `SET_ASSOCIATION`: impostazione di un'associazione tra hardware programmabile e comando di controllo.
- comandi per `thmx_usb_send_ra_req()`:
  - `ISO_IN_DBG`: invio da parte del gadget di una stringa di debug sull'endpoint isochronous;
  - `BULK_IN_DBG`: invio da parte del gadget di una stringa di debug sull'endpoint bulk;

– `CHECK_ISO_IN_STATUS`: controllo da parte del gadget dello stato di invio dei campioni audio, con conseguente invio di una stringa di rapporto sull'endpoint bulk (cfr. 6.2.2).

Tra i comandi associati alla prima funzione, in realtà, gli ultimi due (`SET_MIDI` e `SET_ASSOCIATION`) non possono essere considerati veri e propri comandi di controllo, in quanto non possono essere associati ad un elemento hardware programmabile. In ogni caso, sono stati comunque inseriti perché il tipo di comunicazione che richiedono è del tutto identica a quella usata per i comandi di controllo.
Sul gadget, i comandi di controllo di qualsiasi tipo vengono ricevuti con la callback relativa all'endpoint di input di tipo interrupt generico (funzione `int_out_complete()` nel file *gadget/thmx_gadget_usb.c*). Questo semplicemente prende il buffer ricevuto e risveglia il thread interno all'oggetto `thmx_gadget`, che si occupa di effettuare il parsing di questa richiesta ed eseguire la funzione appropriata.
Nello user-space del sistema host, i comandi di controllo vengono eseguiti tramite il comando `thmx_exec_ctl_command`, invocabile da un utente qualsiasi grazie al fatto che è stato impostato il SUID su di esso e, quindi, riesce ad aprire il file */dev/thereminux* in scrittura per i comandi ioctl.

6.2.7 Il driver UART

Lo sviluppo del driver UART usato per la scrittura dei dati di MIDI-out ha rappresentato, forse, la fase di implementazione in cui ci si è dovuti interfacciare a più basso livello con i dispositivi hardware veri e propri. Infatti, anche se l'API UART (definita nel kernel in `<linux/serial_core.h>`) rappresenta già di per sé un'astrazione che nasconde i dettagli del driver *TTY* sottostante, c'è comunque il bisogno, per la scrittura dei dati e per il setup dettagliato della porta, di avere a che fare con i registri fisici trovati sull'hardware. Per questo, ci si è ispirati al driver implementato nel kernel (2.6.37) in *drivers/serial/omap-serial.c* e, soprattutto, alla documentazione tecnica per la programmazione dei processori della serie OMAP35x [15]. Il driver implementato si trova nel file *gadget/thmx_gadget_uart.c*.
In fase di inizializzazione (funzione `gthmx_uart_init()`) si allocano e registrano le due strutture fondamentali del sistema, cioè la `struct uart_driver` e la `struct uart_port`. L'inizializzazione di quest'ultima è la parte più interessante di questa fase:

```
/*uart port configure*/
uport = &u_data->port;
uport->ops = &gthmx_uart_ops;
```

```
spin_lock_init(&uport->lock);
uport->line = 0;
uport->dev = &platform->dev;
uport->private_data = gthmx;
uport->mapbase = OMAP_UART3_BASE;
uport->membase = ioremap_nocache(uport->mapbase, GTHMX_UART_SIZE);
if (!uport->membase) {
   printk(KERN_WARNING "thmx_gadget: cannot ioremap
                        the mapbase address");
   goto err_plat_dev;
}
uport->iotype = UPIO_MEM;
uport->flags |= UPF_BOOT_AUTOCONF;
uport->irq = GTHMX_UART3_IRQ;
uport->uartclk = OMAP16XX_BASE_BAUD * 16;
uport->regshift = 2;
uport->fifosize = GTHMX_UART_FIFO_SIZE;

ret = uart_add_one_port(uart, uport);
...
```

Dato il basso livello a cui ci si interfaccia, la porta UART viene associata ad un generico `platform_device` appositamente allocato, che viene utilizzato soltanto per l'assegnamento al dispositivo della porta (`uport->dev = &platform->dev`). In realtà, le risorse fisiche della piattaforma hardware sono state definite staticamente in base alla documentazione tecnica [15]: in questo modo, si è trovata la locazione dove la porta `UART3` (quella che si interfaccia al connettore RS232 sulla Beagle Board) è mappata in memoria fisica, cioè `OMAP_UART3_BASE = 0x49020000`. Da qui, la zona di memoria utilizzata per le porte di I/O non può essere trattata allo stesso modo delle normali zone in RAM: ad esempio, per queste, è importante che venga mantenuto strettamente l'ordine delle operazioni che vengono specificate nel driver, evitando possibili ottimizzazioni da parte del compilatore che potrebbero portare a cambiare l'ordine delle scritture sulle linee fisiche. Inoltre, la memoria di I/O rappresenta la zona di memoria messa a disposizione proprio dal chip interessato: in questo senso, esistono zone dedicate ai trasferimenti audio, ethernet, di dati grafici, ecc. Tra questi, il processore OMAP3550 usa una zona di I/O relativa alla porta UART che dobbiamo usare. Per ottenerla, per prima cosa dobbiamo effettuare una "traduzione" dell'indirizzo di memoria fisica di cui disponiamo verso una area di memoria virtuale che permetta l'accesso a tale zona di I/O, e questo è fatto tramite la `ioremap_nocache()` (usata qui la

versione `_nocache` perché, nella stessa area di I/O sono presenti registri per la scrittura dei dati e per la configurazione dell'hardware, per cui non è desiderabile il caching delle operazioni). Successivamente, come mostrato nella funzione `gthmx_uart_request_port()` richiamata dal kernel sempre in fase di inizializzazione, si utilizza la `request_mem_region()` per ottenere tutta l'area di memoria da indirizzare per usare la porta UART3. Infine, si nota che è stato specificato il clock della porta come viene specificato in [15], e usando una macro che è stata ripresa dal file *arch/arm/mach-omap1/serial.c*. Successivamente, una volta chiamata la funzione `uart_add_one_port()`, il kernel richiama alcuni metodi di inizializzazione definiti dal driver stesso per il setup dell'hardware. Tra questi c'è la `gthmx_uart_request_port()` già vista ma, più importante, la `gthmx_uart_set_termios()`, che si occupa proprio della scrittura fisica nei registri per impostare le caratteristiche del protocollo di comunicazione. Per sfruttare questi registri, è stata utilizzata la seguente procedura:

1. viene visto, dal manuale tecnico [15], il nome del registro da utilizzare e si trova il suo nome nell'header `<linux/serial_reg.h>`;

2. viene capito quali sono i bit da scrivere su quel registro per ottenere l'effetto desiderato;

3. vengono letti i valori esadecimali specificati dalle macro, contenute in `<linux/serial_reg.h>`, specifiche per quel registro;

4. vengono tradotti quei valori esadecimale in binario;

5. viene scelta la macro che scrive sui bit che serve siano attivati.

Chiaramente, per la fase di studio, partendo dal codice di esempio del driver `omap-serial` si è proceduto in maniera pressoché inversa, partendo dal nome della macro che veniva usata, andando a trovare a quale registro si riferiva e quali erano i bit che scriveva, così da capirne il comportamento. Ad esempio, nel codice si scrive nel registro `UART_LCR` (Line Control Register) il valore `UART_LCR_DLAB` (LCR Divisor Latch Access Bit). Sul file di header `<linux/serial_reg.h>` si trova la macro `UART_LCR_DLAB = 0x80`. In binario questa corrisponde a `10000000`. Sul manuale tecnico [15] si va a cercare il LCR e si vede, nella specifica del bitwise, che scrivere `0x1` sul bit 7 (la numerazione parte da 0) significa accedere al bit `DIV_EN` e impostare il Divisor Latch Enable, ovvero scegliere l'operational mode del processore [15].
In questo modo, è stato possibile sfrondare molte delle impostazioni che non servivano alla comunicazione di MIDI-out e specificare quelle necessarie. Tra

queste, è stato necessario impostare il Baud Rate della comunicazione. Per programmarlo dinamicamente sul processore OMAP, come si legge in [15], è necessario specificare un *divisore* che verrà applicato al clock della UART, così da ottenere la velocità desiderata. Nel paragrafo 17.1.1 del manuale tecnico [15] si legge che, considerando che il clock funzionale della UART è a 48 MHz e specificando un oversampling 16 sul clock di base, la formula per calcolare il baud rate da programmare è:

```
Baud rate = (functional clock/16)/div
```

da cui si ottiene il valore del divisore

```
31250 = (48000000/16)/div =>
div = ((48000000/16)/31250) = 96
```

Oltre a questo, è stato necessario specificare 8 bit di dati usati per la comunicazione (`UART_LCR_WLEN8`), 1 stop bit (`~UART_LCR_STOP`) e nessun bit di parità (`~UART_LCR_PARITY`).

6.2.8 La gestione dell'hardware programmabile

La creazione del driver per il pulsante programmabile della Beagle Board (implementato nel file *gadget/thmx_ gadget_ proghw.c*) ha richiesto che ci si interfacciasse all'API GPIO definita in `<linux/gpio.h>`. Per questa, anche seguendo la documentazione nel kernel trovata in *Documentation/gpio.txt*, esistono delle semplici funzioni per registrare la linea GPIO (tramite la `gpio_request()`) e impostarla come output o input, il nostro caso (`gpio_direction_input()`). Fatto questo, è possibile ricavare la linea di irq relativa al GPIO registrato (`gpio_to_irq()`) e associarvi un interrupt handler tramite la `request_irq()`. All'interno di questa funzione è stato specificato di avviare l'interrupt soltanto sul fronte di salita del segnale (ovvero quando il pulsante è premuto) tramite la macro `IRQF_TRIGGER_RISING` passata tra i flag di `request_irq()`.
Oltre a questo, l'esecuzione dell'handler è lasciata al `thmx_gadget_aux_thread` che, quando la sua routine viene risvegliata, legge il comando di controllo correntemente impostato per l'hardware al quale lui è associato e richiama la `gthmx_exec_ctl_cmd()` (cfr. 6.2.6).

6.2.9 La disconnessione dello strumento

In ultimo, quando lo strumento viene disconnesso fisicamente dal sistema o scollegato come modulo, avviene la pulizia sia lato gadget sia lato host. Come spiegato nelle sequenze mostrate nel paragrafo 5.4, questo task

essenzialmente avviene richiamando una per una le funzioni di pulizia definite dai singoli oggetti contenuti nell'oggetto principale (`thmx` e `thmx_gadget` rispettivamente), per poi de-registrare il modulo presso il kernel. Essendo coivolti diversi kernel thread per l'esecuzione, questo ha richiesto una sincronizzazione per la terminazione di tali thread. Tale sincronizzazione è stata implementata impostando una opportuna variabile atomica da ambo i lati (`thmx.disconnect_state` sull'host e `thmx_gadget.running` sul gadget). Visto che tutti i thread eseguono dei loop infiniti in cui rimangono in attesa su delle apposite `wait_queue` finché il chiamante non li sveglia, e solo allora eseguono il proprio lavoro per poi tornare in attesa, è stato sufficiente inserire dei controlli ogni volta che i thread escono dall'attesa: se è stata impostata la variabile di disconnessione del sistema, l'esecuzione può terminare. In questo modo, il thread principale semplicemente imposta tale variabile e poi sveglia il thread. Infine, però, la sincronizzazione è stata forzata utilizzando delle `struct completion`: ogni volta che il thread "padre" risveglia un figlio per la terminazione, utilizza la funzione `wake_up_interruptible_sync()` la quale fa sì che il processo chiamante mantenga la cpu anche dopo la chiamata a tale funzione (che comunque non è atomica). Questo per evitare che il thread figlio acquisisca il processore prima che il thread chiamante non lo stia aspettando per la terminazione. Infatti, subito dopo la chiamata a `wake_up_interruptible_sync()`, il thread chiamante esegue la `wait_for_completion()`, attendendo così la terminazione del thread svegliato. Questo, una volta risvegliato e dopo aver notato che il modulo si trova in uno stato di terminazione, prima di uscire chiama la `complete()`, che risveglia il processo in attesa di tale terminazione.

Capitolo 7

Test effettuati e risultati ottenuti

Il sistema in esecuzione è mostrato in Figura 7.1.

Figura 7.1: Il setup del Thereminux in esecuzione

Come si nota, il sistema host è stato collegato alla Beagle Board tramite cavo USB *Mini-AB*, che permette anche la funzionalità OTG (cfr. 6.1.2). Inoltre, La Beagle Board mette a disposizione la sua porta RS232 tramite un header

hardware a 10 pin [16], per cui è stato necessario implementare un cavo per convertire questo connettore in un classico connettore seriale a 9 pin. Da questo, tramite un cavo *Null Modem* [68], il cavo seriale è stato collegato ad un convertitore RS232-USB che ha permesso il collegamento seriale con l'host.
Il gadget è stato monitorato tramite il comando *minicom*, in cui la comunicazione è stata impostata, seguendo la guida su [39], con un Baud Rate a 115200 bps, 8 bit di dati, 1 stop bit e nessun bit di parità.
La scheda audio Audiophile esterna (cfr. 6.1.1), inoltre, è stata anch'essa collegata tramite USB all'host, permettendo di avviare il sistema jack.
Sono stati effettuati diversi test approfonditi per la validazione del lavoro svolto. Nel seguito verranno descritti dettagliatamente.

7.1 Test sulla trasmissione di dati audio

Per prima cosa, è stata validata la connessione audio tra host e gadget procedendo a verificare che i dati audio inviati dal gadget fossero corretti. Per fare questo, è stato implementato il metodo `read()` accessibile dal nodo */dev/thereminux*, il quale semplicemente effettua le letture sulla `kfifo()`, contenente i campioni audio letti dal thread isochronous, al posto del modulo `thmx_vtm` (cfr. 6.2.2 e 6.2.5). Per fare questo, in ogni caso, è stata prevista una macro, chiamata `THMX_AUDIO_DEBUG` (nel file *host/headers/thmx.h*) che, quando definita, fa sì che il thread di ricezione sull'endpoint isochronous non chiami la conversione Voltage-To-MIDI sui campioni letti (evidando così i possibili conflitti di lettura dalla `kfifo`) e che la funzione `thmx_read()` effettivamente effettui la lettura di questi dati. Infatti, quando la macro non viene definita, tale funzione ritorna allo user space con un `-EINVAL` e stampa un messaggio comunicando l'impossibilità di eseguire l'operazione. In ogni caso, in fase di debug questa funzionalità è stata utilizzata con il programma in user space `thmx_debug_get_audio` (eseguibile da un utente privilegiato, cfr. 5.3.2) che esegue un loop di letture sul dispositivo e salva il risultato nel file *~/Scrivania/thmx_audio.raw*. L'applicazione tiene conto del fatto che la `thmx_read()` ritorna il valore `ERESTART` quando la fifo è vuota nel momento in cui si cerca di leggerla. In questo caso, il programma effettua un polling di letture finché non arrivano dati validi e, a quel punto, li salva sul file di output. Una volta salvato, tale file è stato poi aperto con un comune programma audio per Linux (è stato usato l'editor di file audio *Audacity* [69]), il quale, dando in input i parametri di descrizione dello stream audio (`S16_LE`, cfr 6.2.5), è riuscito ad interpretare correttamente tale stream e a suonarlo come normale file audio (Figura 7.2).

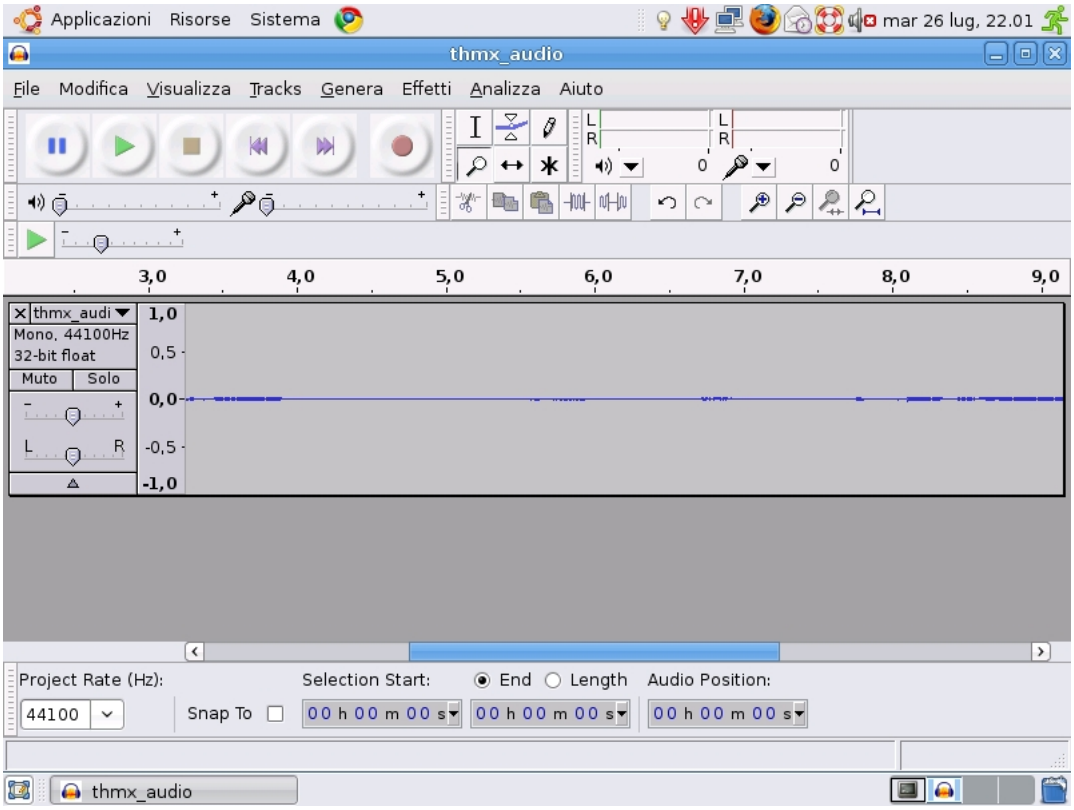

Figura 7.2: Il file di dati audio grezzi aperto con l'applicazione *Audacity*

Come vediamo, il segnale si presenta piuttosto basso come ampiezza, ma questo è chiaro visto che è stato inviato tramite un microfono direttamente collegato alla Beagle Board, senza amplificazione. In ogni caso, l'esecuzione del file mostra che ciò che viene inviato dal microfono è ricevuto correttamente.
Per ottenere una bassa latenza, si è visto che è sufficiente impostare un numero di pacchetti inviati con le richieste isochronous pari a 4 e un intervallo di polling delle richieste (il parametro `urb->interval` di cui si è parlato nel paragrafo 6.2.2) pari a 8. Con valori superiori di intervallo si è notata una crescente latenza ottenuta, mentre si è notata una certa instabilità nel sistema se si impostano richieste troppo stringenti in termini di tempo (ad esempio, un intervallo di 1 con 32 pacchetti). Per farsi un'idea dei motivi di questo fatto, si è provato ad impostare tali richieste di alte prestazioni disabilitando l'uso della conversione Voltage-To-MIDI (e, quindi, della FPU) e si è notato che le instabilità sparivano, riuscendo tranquillamente a sostenere le trasmissioni. Si suppone, allora, che l'uso della FPU, richiedendo la disabilitazione

della *preemption* con la chiamata, da parte di `kernel_fpu_begin()`, della funzione `preempt_disable()` e, oltre a questo, effettuando il salvataggio e ripristino dello stato dei registri della FPU ad ogni ciclo di esecuzione, influisca notevolmente sulle prestazioni generali del sistema, costringendo a non "sforzare" troppo. In ogni caso, l'impostazione dei parametri come descritti pocanzi ha permesso di ottenere dei valori di latenza del tutto accettabili, per lo meno per quanto riguarda la risposta del sistema "a orecchio".

7.2 Far suonare il Thereminux

Il test su come lo strumento "suona" è stato fatto collegandolo dopo aver inizializzato il sistema audio con jack pronto e funzionante. Una volta che il gadget si è attivato, inizia immediatamente a inviare campioni audio, che vengono tradotti in messaggi MIDI e inviati al livello applicativo. Qualora, in fase di avvio, il gadget ritarda troppo ad iniziare l'invio di dati rispetto alle richieste dell'host, causando la mancata ricezione di pacchetti iniziali, subito l'host invia una richiesta di ripristino e riceve dal gadget la stringa `ISOCH_IN_STOPPED` (che dimostra, semplicemente, che il gadget ancora non ha avuto il tempo di iniziare le trasmissioni).
Una volta collegato e attivo, lo strumento appare nell'interfaccia di jack-control con i due dispositivi MIDI allocati (di default, il profilo di configurazione inviato dal gadget abilita anche il MIDI-out). La schermata mostrata in Figura 7.3 evidenzia quanto detto.
Nella Figura sono mostrati alcuni aspetti salienti dell'esecuzione: sulla destra, una console mostra il collegamento seriale con il gadget, il quale si trova in uno stato di invio e resta fermo sul messaggio "verboso" stampato dal `thmx_gadget_alsa_capture`. All'interno dell'editor di connessioni di jack-control, invece, sono mostrati i due dispositivi MIDI, uno di input (tra i *Readable Clients*) e uno di output (tra i *Writable Clients*). In particolare, la porta di MIDI-in è stata collegata ad un sintetizzatore audio (il *Fluid Synth*) che è in grado di suonare i messaggi MIDI che riceve, inviandoli alla scheda audio. In questo modo, è possibile applicare svariati suoni ai messaggi prodotti dal Thereminux (di default, il sintetizzatore utilizza il suono di un pianoforte). Poiché il profilo di configurazione è impostato inizialmente sul "glissando", le note suonate vengono collegate tra loro, come era l'effetto desiderato.
Certo, salta all'orecchio una certa inaccuratezza della conversione delle note "cantate" in note MIDI, che spesso si avvicinano a quanto inviato attraverso il microfono, ma spesso non rappresentano precisamente le nota suonate. I problemi, in questo caso, sono sicuramente diversi: per prima cosa, il segna-

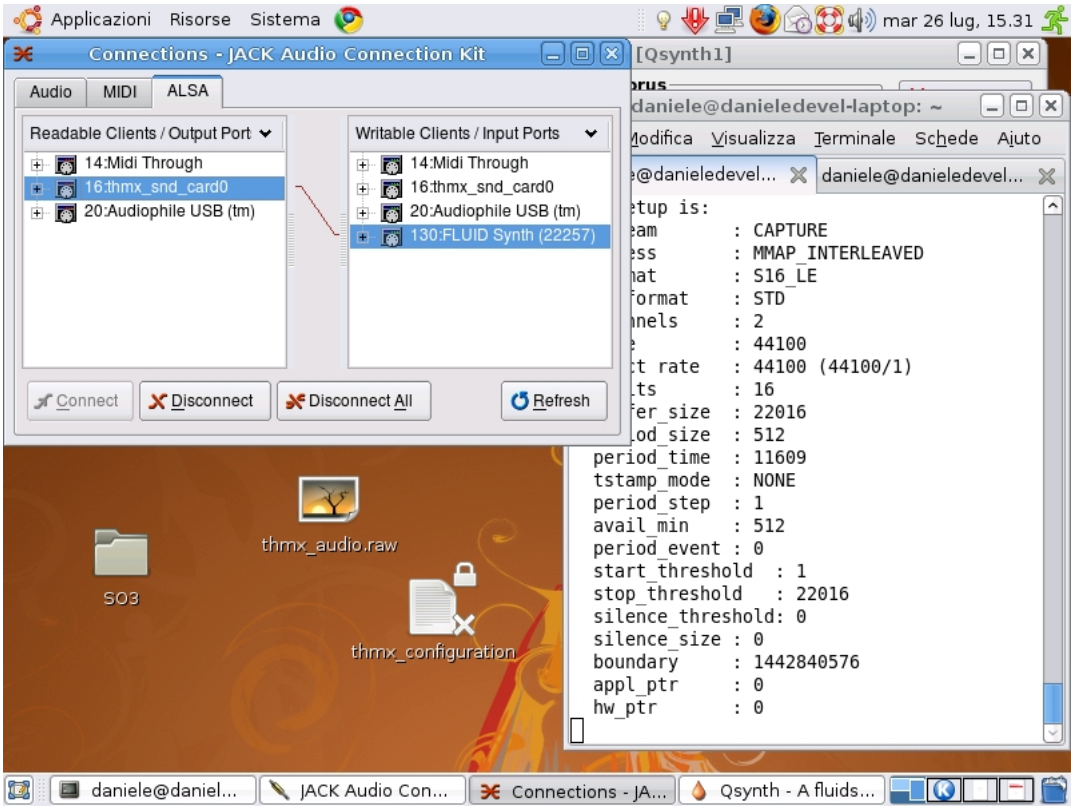

Figura 7.3: Uno screenshot di utilizzo del Thereminux

le vocale inviato dal microfono è piuttosto difficile da analizzare, essendo la voce umana estremamente ricca di componenti armoniche e inarmoniche che rendono ardua l'estrazione di una sinusoide fondamentale; inoltre, il segnale viene ricevuto dallo strumento con un volume piuttosto basso, come mostrato anche nel paragrafo precedente (cfr. 7.1). Ancora, per evitare di ascoltare un flusso velocissimo di note che si susseguono una dietro l'altra, è stato creato, dentro `thmx_aux_thread`, un contatore che consente l'invio dei messaggi al livello applicativo soltanto una volta ogni 5 buffer ricevuti. Anche questo può causare qualche leggero "spostamento" delle note di output rispetto a quelle correntemente inviate. Infine, come trattato anche nella tesi [13] in cui si è cercato di sviluppare un controller MIDI basato sulla voce confrontando diversi metodi e librerie, i risultati sperimentali della pitch detection della libreria *aubio*, presi da soli, sono spesso non troppo accurati. In quella tesi, infatti, visti gli scarsi risultati dei metodi soltanto basati su pitch detection, si è deciso di adoperare un approccio più complesso, in cui i flussi audio sono prima sottoposti ad un algoritmo di *onset detection* (riconoscimento della

presenza o meno di una nota nel segnale) e l'output di questo è poi usato per la pitch detection. Dunque, sembra che sarebbe necessario un un ulteriore approfondimento di queste tecniche di manipolazione di segnali audio, rimanendo la questione dell'estrazione dei valori di pitch da tali segnali uno dei problemi aperti all'interno del *Digital Signal Processing*.

7.3 Test dei comandi di controllo

Ognuno dei comandi di controllo è stato testato ed è risultato funzionante. Per cominciare, una volta collegato lo strumento, il file di configurazione viene creato sul desktop (cfr. Figura 7.3), non leggibile né scrivibile dall'utente comune. Tramite il comando `thmx_read_profile`, si ottiene l'output mostrato in Figura 7.4.

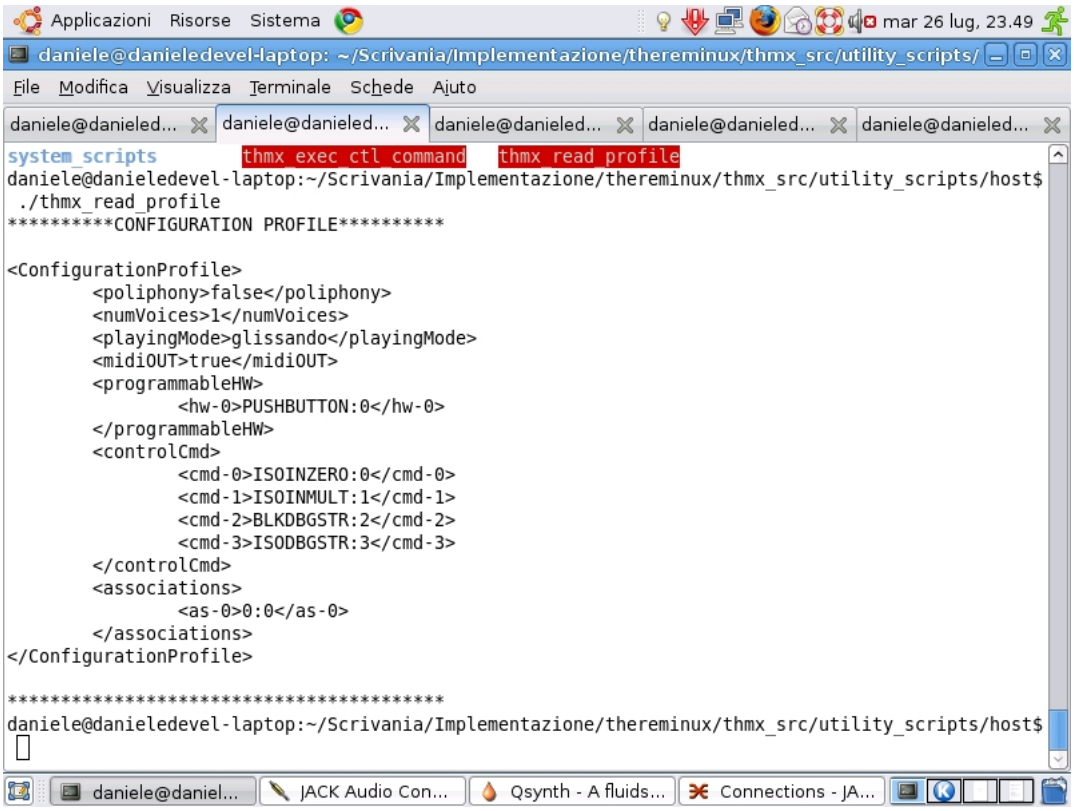

Figura 7.4: Esecuzione del comando `thmx_read_profile`

La formattazione rispetta quella con cui effettivamente è salvato il file del profilo di configurazione. Inoltre, se l'utente vuole avere un promemoria soltanto delle stringhe da utilizzare per inviare comandi di controllo, può utilizzare il

comando `thmx_read_ctl_commands`. L'output è una semplice stampa di tali stringhe (Figura 7.5)

```
daniele@danieledevel-laptop:~/Scrivania,
 ./thmx_read_ctl_commands
THEREMINUX AVAILABLE COMMANDS:
command 0: ISOINZERO
command 1: ISOINMULT
command 2: BLKDBGSTR
command 3: ISODBGSTR
```

Figura 7.5: Esecuzione del comando `thmx_read_ctl_commands`

Successivamente, per eseguire effettivamente tali comandi di controllo, è possibile usare il comando `thmx_exec_ctl_command`. In Figura 7.6 viene mostrato il messaggio stampato sul gadget alla ricezione del comando `BULK_IN_DBG` (cfr. 5.3.4).

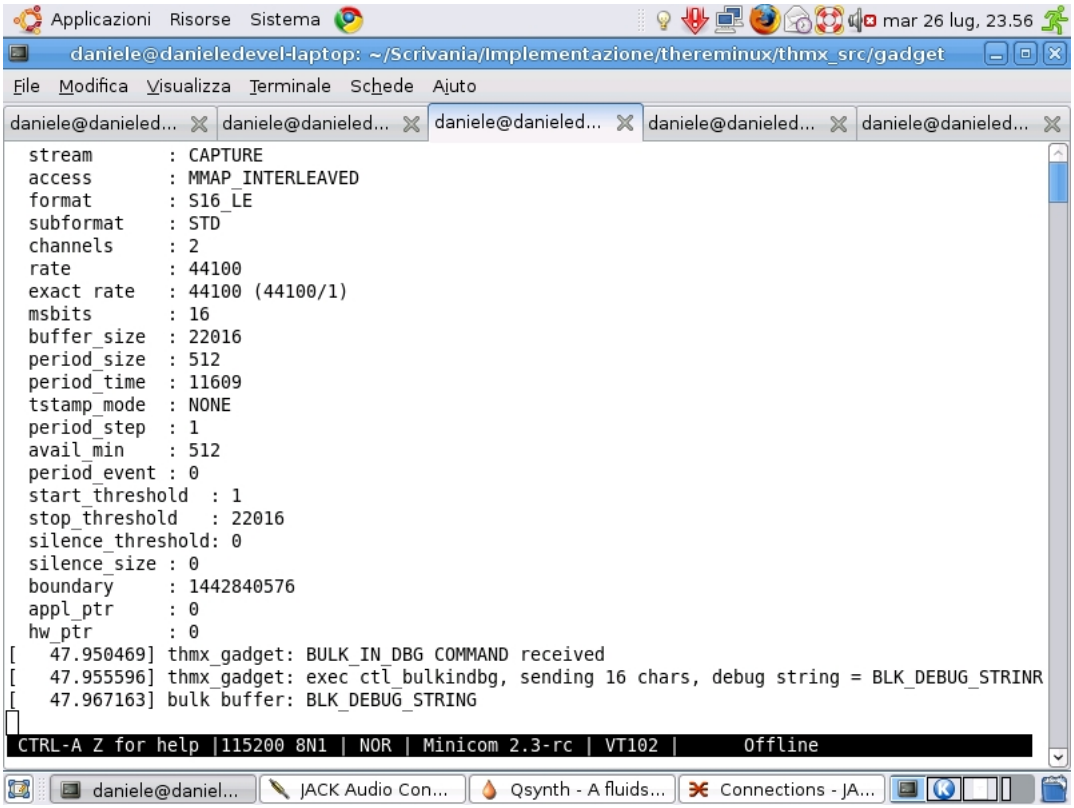

Figura 7.6: Esecuzione del comando di controllo `BULK_IN_DBG` sul gadget

In maniera simile, è stato verificato il funzionamento di tutti gli altri comandi di controllo. Particolarmente interessante è il comando di impostazione del

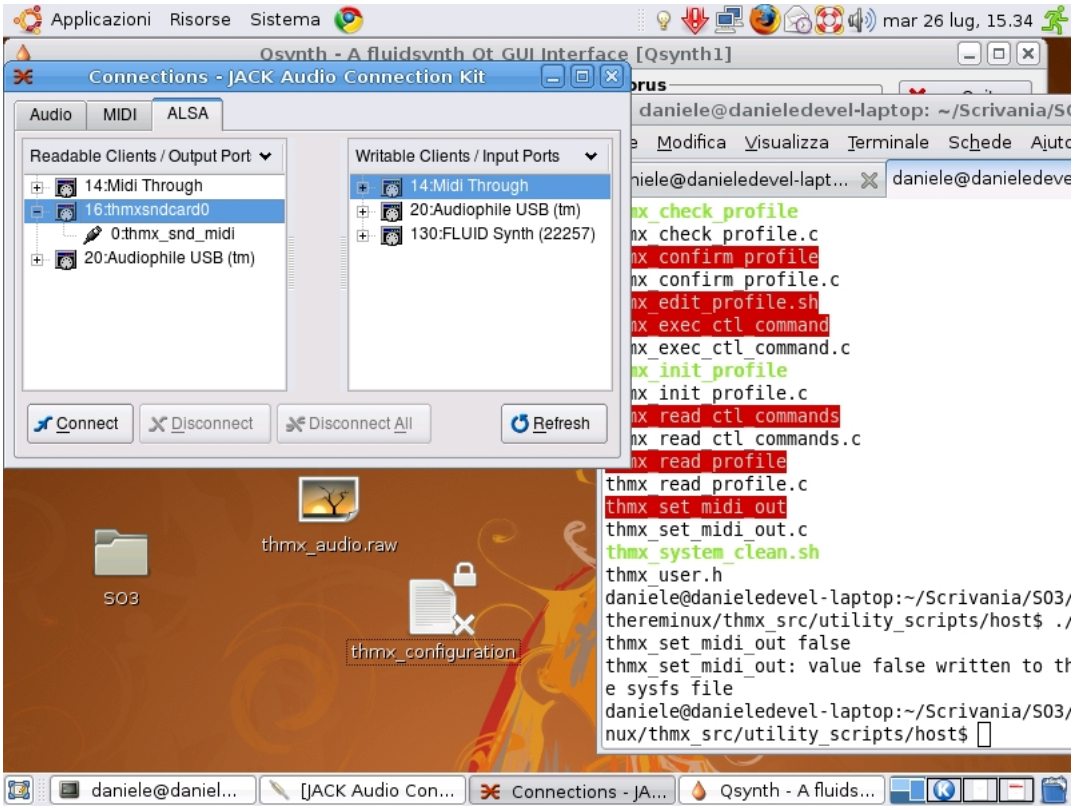

Figura 7.7: Risultato dell'esecuzione della rimozione dinamica del MIDI-out

MIDI-out dinamicamente. Essendo questo di default attivato, per prima cosa è stato invocato il comando `thmx_set_midi_out false`. Questo ha portato alla disconnessione e ricreazione dell'interfaccia MIDI dello strumento, portando al risultato mostrato in Figura 7.7.
Qui, come vediamo, non è più presente la porta di MIDI-out tra i *Writable Clients* di jack-control, e sulla destra si visualizza l'output del comando ("value false written to the sysfs file").

7.4 Test del MIDI-out

L'interfaccia hardware di MIDI-out, in realtà, è stata sviluppata interamente, dando però dei problemi in fase di esecuzione, perché le note non riescono ad arrivare correttamente all'interfaccia di MIDI-in della scheda audio Audiophile. La verifica della parte software è stata effettuata in maniera esaustiva, arrivando alla conclusione che sono necessari degli strumenti di debug hardware più avanzati di quelli in possesso (come, ad esempio, un oscilloscopio).

Per verificare il lato software, sono stati effettuati test sui vari parametri impostati per la connessione seriale:

- per prima cosa, si è tentato di collegare il sistema normalmente al monitor seriale dell'host (tramite *minicom*): come detto, questo lavora ad un baud rate di 115200 bps, 8 bit di dati, 1 stop bit e nessun bit di parità. In pratica, sono le stesse impostazioni del MIDI eccezion fatta per il baud rate. Allora, per prima cosa è stato modificato il divisore per l'impostazione della UART (cfr. 6.2.7) così da mandarlo a 115200 bps, e la comunicazione avviene correttamente;

- successivamente, si è provato a vedere se funziona la reimpostazione del baud rate tramite la tecnica del divisore descritta nel paragrafo 6.2.7. Per verificarlo, si doveva provare a vedere se si riusciva ad ottenere una connessione a *minicom* con un altro baud rate specificato. Poiché il MIDI ha una velocità di 31250 bps, non standard rispetto al set di velocità disponibili per la UART, si è provato con una velocità che si avvicina a questa, ovvero 38400 bps. Dunque, si è impostato *minicom* per lavorare a questa velocità, così come anche il driver sul gadget. In questo modo, avviene la seguente cosa: la fase di avvio della Beagle Board (controllata dal boot loader ed impostata di default a 115200 bps) non viene interpretata correttamente da *minicom* e risulta in una serie di caratteri illeggibili, fino ad arrivare all'inizializzazione di `thmx_gadget`, che riconfigura la velocità della porta e fa sì che quanto stampato successivamente si legga correttamente;

- in ultimo, si è verificato che le scritture effettivamente avvenissero. Per fare questo si è reimpostato il baud rate a 115200, così da permettere un debug tramite *minicom*, e si è iniziato a inoltrare quanto arrivava sulla porta di MIDI-in del Thereminux alla sua porta di MIDI-out. Questo ha fatto sì che i messaggi prodotti venissero direttamente inoltrati al gadget e che questo li scrivesse sulla linea seriale. Il risultato è mostrato in Figura 7.8.
  Sulla destra, vediamo l'editor di connessioni di jack-control. Qui la porta di MIDI-in è stata collegata, oltre che con il solito sintetizzatore software, anche con il MIDI-out dello strumento. Sulla sinistra, vediamo la finestra che mostra i dati scritti sulla seriale dallo strumento che vengono riportati da *minicom* come una serie di byte effettivamente inviati. Il fatto che questi byte siano visualizzati in parte illeggibili nella finestra di *minicom*, pur essendo il baud rate quello corretto, è dovuto al fatto che i valori esadecimali che rappresentano i byte dei messaggi MIDI non sempre sono tra quelli inclusi all'interno dei caratteri ASCII

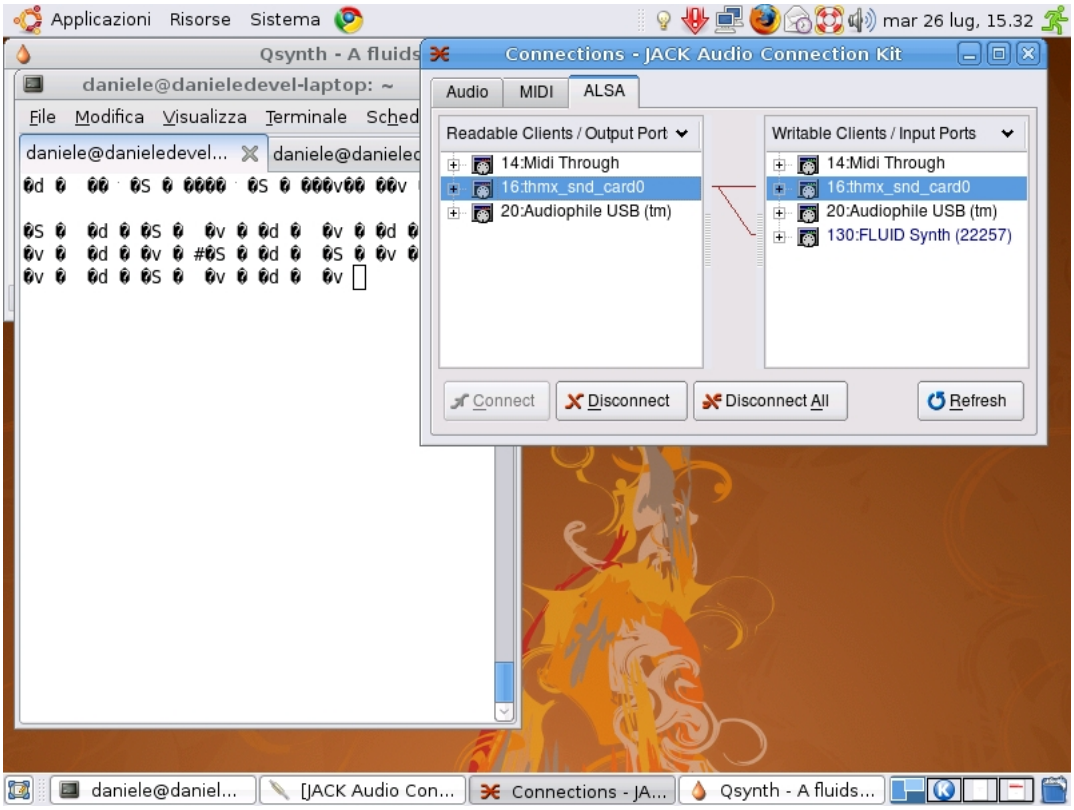

Figura 7.8: Risultato della scrittura di messaggi MIDI sulla porta di MIDI-out

standard. Quando questo avviene, vediamo delle lettere comparire sulla console (cosa che non avviene quando la trasmissione è impostata ad un baud rate sbagliato).

Dunque, la conclusione tratta è stata che il problema debba necessariamente trovarsi nel convertitore RS232-to-MIDI sviluppato. Dopo aver effettuato tutti i test statici del caso (usando un normale tester per verificare i valori di voltaggio nei vari punti del circuito, così come inviando segnali fissi di voltaggio in input del circuito e vedendo l'output) che hanno dato risultati positivi, si è deciso di rimandare ad un momento futuro in cui si avrà a disposizione un oscilloscopio o un *logic analyser* con il quale poter confrontare i segnali in uscita da dispositivi differenti. Per esempio, la scheda Audiophile ha integrata una porta di MIDI-out che invia segnali correttamente. Per verificarlo, è stato collegata in loopback con la porta di MIDI-in della stessa scheda e, poi, i messaggi prodotti dal Thereminux sono stati inoltrati tramite jack-control su tale MIDI-out hardware. I valori ottenuti sul MIDI-in della Audiophile,

quindi, sono stati inviati al sintetizzatore software, con il risultato le note sono state suonate correttamente. Per questo, con l'ausilio di un oscilloscopio sarà possibile acquisire un campione prodotto da tale porta di MIDI-out e confrontarlo con quello prodotto dalla scheda sviluppata. Questo permetterà di ottenere dati rilevanti sul segnale che viene generato da tale scheda.

Capitolo 8

Conclusioni: possibili miglioramenti e integrazioni

Sebbene il lavoro abbia richiesto un tempo maggiore di quello pensato all'inizio, i risultati sono sembrati piuttosto soddisfacenti. Naturalmente, nei capitoli precedenti sono stati trattati alcuni aspetti, riguardanti l'uso dell'applicazione finale implementata, che hanno bisogno di alcune migliorie. Tra questi, l'accuratezza della conversione delle note ricevute in input è un punto importante da sviluppare e che, probabilmente, richiederà l'ausilio di tecniche più avanzate (coma la onset detection), come spiegato nel paragrafo 7.2. Inoltre, la possibilità di utilizzare, come era stato auspicato all'inizio, uno strumento come il Theremin in grado di generare delle sinusoidi pure potrebbe semplificare di molto il processo di riconoscimento delle note, essendo la voce umana (così come la maggior parte degli strumenti acustici) molto complessa nella sua composizione armonica. Infine, l'invio di note di MIDI-out attraverso lo strumento è stato implementato in software, ma ancora manca l'interfaccia hardware per il collegamento fisico dello strumento creato con altri dispositivi.
Detto ciò, il sistema sviluppato ha rappresentato un esercizio importante di implementazione *kernel side*, permettendo di esplorare svariati sotto-sistemi e API messe a disposizione del kernel. Dato il personale interesse dell'autore per lo sviluppo a basso livello, questo lavoro resterà una risorsa importante a cui attingere anche nella prosecuzione della carriera di studio e professionale.

Bibliografia

[1] P. Horowitz, W. Hill, *The Art of Electronics*, Cambridge University Press, 1989

[2] AA.VV., *Universal Serial Bus Specification - rev. 2.0*, 2000

[3] *Ivi*, cap. 9, pp. 239 - 274

[4] J. Corbet, A. Rubini, G. Kroah-Hartman, *Linux Device Drivers - third edition*, O'Reilly ed., 2005

[5] D.P. Bovet, M. Cesati, *Understanding the Linux Kernel, third edition*, O'Reilly ed., 2005

[6] T. Iwai, *Writing an ALSA Driver*, documento open-source scaricabile da http://www.kernel.org/pub/linux/kernel/people/tiwai/docs/writing-an-alsa-driver.pdf, 2005

[7] D. Gerhard, *Pitch Extraction and Fundamental Frequency: History and Current Techniques*, technical report, Dept. of Computer Science, University of Regina, 2003.

[8] A. de Cheveign, H. Kawahara, *Yin, a fundamental frequency estimator for speech and music*, Journal of the Acoustical Society of America, 111(4), 2002

[9] M. Noll, *Cepstrum pitch determination*, The Journal of the Acoustical Society of America 41 (2): 293 - 309, 1967

[10] M. Noll, *Pitch determination of human speech by the harmonic product spectrum, the harmonic sum spectrum, and a maximum likelihood estimate*, Proceedings of the Symposium on Computer Processing ing Communications: 779 - 97, 1969

[11] F. Bach, M. Jordan, *Discriminative training of Hidden Markov Models for multiple pitch tracking*, Proceedings of the International Conference on Acoustics, Speech, and Signal Processing, 2005

[12] E. Barnard, R.A. Cole, M.P. Vea, F.A. Alleva, *Pitch detection with a neural-net classifier*, IEEE Transactions on Signal Processing 39 (2): 298 - 307, 1991

[13] A. Nagi, *A user configurable voice based controller for Midi synthesizer - A Prototype for Real-Time Performance*, Master Thesis UPF, 2010

[14] Texas Instruments, *OMAP3530/25 Datasheet*, 2009

[15] Texas Instruments, *OMAP35x Applications Processor - Technical Reference Manual*, 2010

[16] beagleboard.org, *BeagleBoard System Reference Manual Rev C4*

Risorse web:

[17] *Musical Instrument Digital Interface*, Wikipedia, http://en.wikipedia.org/wiki/MIDI

[18] *Glissando*, TheFreeDictionary, http://www.thefreedictionary.com/glissando

[19] *Hot Plug*, Linux Journal, http://www.linuxjournal.com/article/5604

[20] *Plug-and-Play-HOWTO*, The Linux Documentation Project, http://tldp.org/HOWTO/Plug-and-Play-HOWTO.html

[21] *Nyquist-Shannon sampling theorem*, Wikipedia, http://en.wikipedia.org/wiki/Nyquist-Shannon_sampling_theorem

[22] *Analog-to-Digital Converter*, Wikipedia, http://en.wikipedia.org/wiki/Analog-to-digital_converter

[23] *Audio Converter*, Wikipedia, http://en.wikipedia.org/wiki/Audio_converter

[24] *Audio frequency*, Wikipedia, http://en.wikipedia.org/wiki/Audio_frequency

[25] *ADC - most popular types*, Hitequest website,
http://www.hitequest.com/Kiss/A_D.htm

[26] *What actually happens when you plug in a USB device?*, Technovelty, weblog of Ian Wienand,
http://www.technovelty.org/code/linux/plugging-in-usb.html

[27] *udev*, wikipedia,
http://en.wikipedia.org/wiki/Udev

[28] *Writing udev rules*, reactivated.net website,
http://www.reactivated.net/writing_udev_rules.html

[29] *MIDI Manufacturers Association*, MMA website,
http://www.midi.org/

[30] *The MIDI Specification*, oktopus.hu website,
http://www.oktopus.hu/imgs/MANAGED/Hangtechnikai_tudastar/The_MIDI_Specification.pdf

[31] *MIDI Note Numbers*, somascape.org website,
http://www.somascape.org/midi/help/notes.html

[32] *ALSA*, official website,
http://www.alsa-project.org

[33] *ALSA unofficial*, unofficial website,
http://alsa.opensrc.org/

[34] *Writing an ALSA driver*, Ben Collins - Technical ramblings of a Linux developer,
http://ben-collins.blogspot.com/2010/04/writing-alsa-driver.html

[35] *Real-time Pitch Detection*, Francois Germain - Schulich School of Music
http://www.music.mcgill.ca/~francois/Pitch_Presentation/Pitch_summary.pdf

[36] *Arduino*, Arduino HomePage
http://www.arduino.cc/

[37] *Teensy USB Development Board*, PJRC: Electronic Projects Components Available Worldwide
http://www.pjrc.com/teensy/index.html

[38] *Beagle Board*, BeagleBoard.org website
http://beagleboard.org/

[39] *Beagle Board eLinux support*, eLinux.org website
http://elinux.org/BeagleBoard

[40] *BeagleBoard.org - X-Loader*, BeagleBoard.org website
http://beagleboard.org/project/X-Loader/

[41] *Das U-Boot - Universal Bootloader*, sourceforge website
http://sourceforge.net/projects/u-boot/

[42] *Real-Time Linux Wiki*, rtwiki webpage,
https://rt.wiki.kernel.org

[43] *A realtime preemption overview*, lwn.net - Linux info from the source,
http://lwn.net/Articles/146861/

[44] *Workqueues get a rework*, lwn.net - Linux info from the source, http://lwn.net/Articles/211279/

[45] *Hyper Low-Latency Audio with a Real-Time Kernel*, youtube video tutorial,
http://www.youtube.com/watch?v=zzvyV1hyDYc

[46] *Kernel Locking Techniques*, Linux Journal website,
http://www.linuxjournal.com/article/5833

[47] *Version 1.3.0 of set_ rlimits*, Jonathan Woithe's Home page,
http://www.physics.adelaide.edu.au/~jwoithe/set_rlimits-1.3.0.tgz

[48] *aubio, a library for audio labelling*, aubio website,
http://aubio.org/

[49] *RS-232 specifications and standard*, Lammert Bies - Computer Interfacing website,
http://www.lammertbies.nl/comm/info/RS-232_specs.html

[50] *Driving Me Nuts - Things You Never Should Do in the Kernel*, Linux Journal,
http://www.linuxjournal.com/article/8110

[51] *Qsynth - Qt GUI Interface for FluidSynth*, Qsynth website,
http://qsynth.sourceforge.net/

[52] *M-Audio website*
http://www.m-audio.com/

[53] *Using the M-Audio Audiophile USB Digital Audio Interface with Linux*
http://people.uleth.ca/~daniel.odonnell/Blog/using-the-m-audio-audiophile-usb-digital-audio-interface-with-linux

[54] *[kernel build]cp .config / oldconfig / xconfig*, LinuxMusicians website
http://www.linuxmusicians.com/viewtopic.php?f=27&t=1008&start=15

[55] *RT PREEMPT HOWTO*, https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO

[56] *Index of /pub/linux/kernel/projects/rt*, http://www.kernel.org/pub/linux/kernel/projects/rt/

[57] *Linux Kernel and Floating Point*, Lifting the Earth using Linux,
http://www.linuxsmiths.com/blog/?p=253

[58] *BeagleBoardLinuxKernel*, http://elinux.org/BeagleBoardLinuxKernel

[59] *Running Real-time distribution on beagleboard*, http://groups.google.com/group/beagleboard/browse_thread/thread/8b56060586cc62e9/cd9016eb812a5c51?lnk=gst&q=real+time&pli=1

[60] *Sourcery G++*, http://www.codesourcery.com/sgpp/lite/arm/portal/subscription?@template=lite

[61] *mkcard.sh v0.5*, http://www.angstrom-distribution.org/demo/beagleboard/mkcard.txt

[62] *Validating Beagle Board (Rev C4 and Rev Bx) with Tests / Diagnostics*, http://code.google.com/p/beagleboard/wiki/BeagleboardRevC3Validation

[63] *Getting Started - OpenEmbedded*, http://wiki.openembedded.net/index.php/Getting_Started

[64] *Useful Target Recipes*, http://docs.openembedded.org/usermanual/usermanual.html#id442079

[65] *Bitbake User Manual* http://bitbake.berlios.de/manual/

[66] *Beagleboard demo files*, http://www.angstrom-distribution.org/demo/beagleboard/

[67] *The libsndfile API*, http://www.mega-nerd.com/libsndfile/api.html

[68] *Null Modem Serial Cable - Null Modem Cables Explained*, http://compnetworking.about.com/od/networkcables/g/bldefnullmodem.htm

[69] *Audacity: Free Audio Editor and Recorder* http://audacity.sourceforge.net/